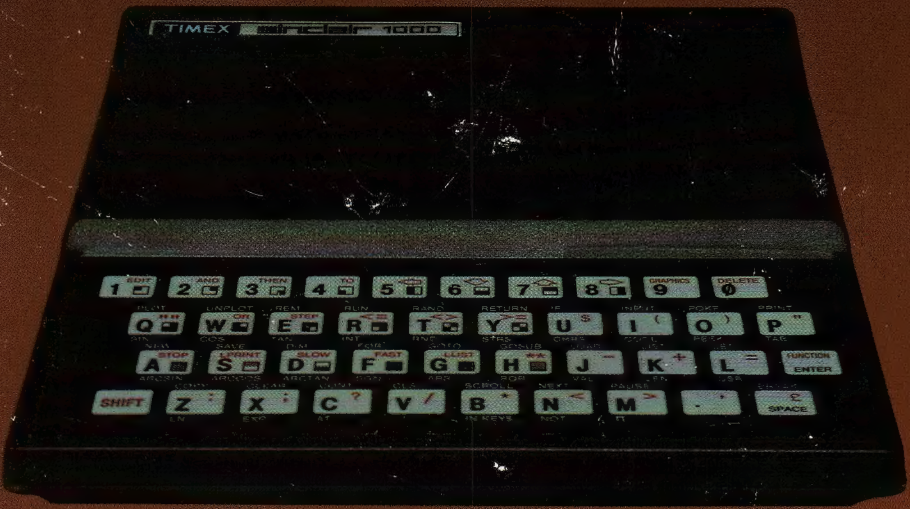
Timex/Sinclair 1000 User's Guide

Volume 2



que

1295

Timex/Sinclair User's Guide

Volume 2

Joseph C. Giarratano

Que Corporation
Indianapolis

This book is dedicated to my family: Jane, Jenna, Melissa, and Anthony.

Library of Congress Catalog No.: 82-42768
ISBN 0-88022-029-5

Cover art: Dick Held
Illustrations: Linda Holman
Photography: Jonathan Mangin
Design: Paul Mangin
Typeset in Helvetica by Alexander Typesetting, Inc.

Manufactured in the United States of America

Published by Que Corporation
7960 Castleway Drive
Indianapolis, Indiana 46250

About the Author

Joseph C. Giarratano received his B.S. and M.S. degrees in physics from California State University at Los Angeles. In 1974 he earned his Ph.D. degree in physics at the University of Texas at Austin.

Dr. Giarratano has worked in the field of medical physics with interests in the applications of computers to medicine. He now works in the area of software development for microprocessor-based products.

Dr. Giarratano is also the author of five other books on computers: *Foundations of Computer Technology, Modern Computer Concepts, BASIC: Fundamental Concepts, BASIC: Advanced Concepts,* and *Timex/Sinclair 1000 User's Guide Volume 1.*

Editorial Director
David F. Noble, Ph.D.

Editor
Diane F. Brown, M.A.

Managing Editor
Paul L. Mangin

Assistant Managing Editor
Tim P. Russell

Table of Contents

Preface

Programs in this Book

Preface

Welcome to Volume 2 of the *Timex/Sinclair 1000 User's Guide*! Now that you've learned the essentials of programming from Volume 1, you're ready for more advanced BASIC programming commands and techniques. In this book, you will see:

- a checkbook balancing program, with cassette data storage
- alphabetical sorting of lists, which you can use to sort record or book collections, recipe cards, names and addresses, etc.
- plotting and drawing
- computer animation and graphics
- video games
- a renumbering program
- how to use machine-language programs

and many other applications. You will also learn the capabilities and limitations of BASIC from the detailed discussions of each program.

The last chapter of this book discusses ways you can expand the hardware and software capabilities of your computer. Since the T/S 1000's introduction to the United States a short time ago, there has been an explosion of new hardware and software products for your computer. Only a sampling of the wide variety of products available is listed here.

After you've mastered the material in Volumes 1 and 2 of the *User's Guide*, you may also want to obtain the *Timex/Sinclair 1000 Pocket Dictionary*. This dictionary serves as a convenient reference to the terms introduced in Volumes 1 and 2 and offers tips on saving memory.

CHAPTER 1
Putting Things in Dimension

Many of your computer's applications may require the handling of much data. For example, you may want to catalog 1,ØØØ records, stamps, recipes, or employee names and payrolls. It is impractical to define a unique variable name for each of the 1,ØØØ items and try to manipulate those variables.

Dimensioned Variables

To handle variables easily, your computer has dimensioned variables. The term *dimensioned* means that you can easily store and retrieve data in the variables by using unique identification numbers. These are like your charge account numbers or house numbers, each identifying a specific account or house. There may be 1,ØØØ houses on Main Street, but there is only one each of the following addresses:

1 Main Street
199 Main Street
852 Main Street

In computer terms, we can write

M(1) to represent the house at 1 Main Street

M(199) to represent the house at 199 Main Street

M(852) to represent the house at 852 Main Street

The M with a number in parentheses is a dimensioned variable. Like the control variable of a **FOR-NEXT** loop, a dimensioned variable must be a single letter. Another term for dimensioned variable is *subscripted* variable because, mathematically, it is written with a subscript following it. For example,

M_1 represents M(1) to the computer

M_{199} represents M(199) to the computer

M_{852} represents M(852) to the computer

The subscripts are written inside parentheses because we can't input a number half a line below the variable name. Another common term is *indexed* variable, for which the variable's unique identification number is used to index or identify it.

Figure 1-1 shows how dimensioned variables are stored in the computer's memory. In this example, the dimensioned variables and the values they contain are

M(1)=2

M(2)=3

M(3)=8.7

M(4)=2.13E14

M(5)= −.ØØ69

The number in parentheses is the index, or subscript. For example, specifying an index or subscript as 3 uniquely identifies the third dimensioned variable. A subscript of 2 specifies the second variable, and so on.

2	3	8.7	2.31E14	−.ØØ69
M(1)	M(2)	M(3)	M(4)	M(5)

2	-3	8.7	2.31E14	-.0069
M(1)	M(2)	M(3)	M(4)	M(5)

Figure 1-1
Numeric dimensioned variables stored in memory

To use dimensioned variables, you need to tell the computer how much memory to reserve for them. This is accomplished through the dimension keyword **DIM**, located over the **D** key.

Let's try a simple program to assign and print dimensioned variables. First, clear the computer's memory and get the K cursor. Now, enter and run the following:

```
 1Ø DIM M(5)
 2Ø LET M(1)=2
 3Ø LET M(2)=3
 4Ø LET M(3)=8.7
 5Ø LET M(4)=2.31E14
 6Ø LET M(5)=-.ØØ69
 7Ø PRINT M(1)
 8Ø PRINT M(2)
 9Ø PRINT M(3)
1ØØ PRINT M(4)
11Ø PRINT M(5)
```

Next, you'll see

```
2
3
8.7
2.31E+14
-.ØØ69
```

This display shows that the values were indeed assigned to the dimensioned variables since they all were printed out.

Looping through Dimensions

The combination of **FOR-NEXT** loops and dimensioned variables makes a powerful tool for manipulating variables. Suppose you have a thousand variables to print out. It would be tiring and boring for you to enter line 70 in the program 200 times. However, computers can do tasks like this, which is one reason you should use a computer. Also, a lot of memory would be used to hold all those **PRINT** commands. Instead, we can change lines 70 through 90 to

```
70 FOR I=1 TO 5
80 PRINT M(I)
90 NEXT I
```

and delete lines 100 and 110. You'll see the same output as before. Now, if you had 1,000 items to print out, you'd only have to change line 70 to

```
70 FOR I=1 TO 1000
```

Obviously, it doesn't matter how many times the FOR-NEXT loop is executed, whether 1 or 1000 times. You can also see how easy it is to index through all the variables, using a **FOR-NEXT** loop. In fact, you can index through any part of the sequence. For example, change line 70 to

```
70 FOR I=3 TO 4
```

and run the program. Now, only M(3) and M(4) are printed.

Also, as with ordinary variables, the values of the dimensioned variables remain after the program stops executing. For example, try

```
PRINT M(3)
```

and you'll see

```
8.7
```

printed at the top of the screen.

Getting Mean

You can easily change the previous program to compute the mean, or average, of the numbers. Try

```
  5 REM MEAN PROGRAM
 10 DIM M(5)
 20 LET M(1)=8
 30 LET M(2)=3
 40 LET M(3)=1
 50 LET M(4)=16.5
 60 LET M(5)=-6.7
 70 LET SUM=0
 80 FOR I=1 TO 5
 90 LET SUM=SUM+M(I)
100 PRINT I;"SUM=";SUM
110 NEXT I
120 PRINT "MEAN=";SUM/5
```

Lines 80 through 110 calculate and print the accumulated SUM each time through the loop. The control variable, I, is also printed so that you can see the loop execute every time. Line 120 prints the mean by dividing the final SUM by the number of variables, 5. When you run this program, you'll see

```
1 SUM=8
2 SUM=11
3 SUM=12
4 SUM=28.5
5 SUM=21.8
MEAN=4.36
```

Instead of altering a program line for each number we want to include, we can easily use an **INPUT**. Try the following version:

```
  5 REM MEAN PROGAM
 1Ø DIM M(5)
 2Ø LET SUM=Ø
 3Ø FOR I=1 TO 5
 4Ø PRINT "NUMBER ";I;"=?";
 5Ø INPUT M(I)
 6Ø PRINT M(I)
 7Ø LET SUM=SUM+M(I)
 8Ø PRINT "SUM=";SUM
 9Ø NEXT I
1ØØ PRINT "MEAN=";SUM/5
```

Now our **FOR-NEXT** loop has been expanded to ask also for input of the data we want both to sum and to average. Line 4Ø asks you to input in line 5Ø the number stored in M(I). Then line 6Ø prints this number back on the screen, confirming that the number has been stored. Lines 7Ø to 1ØØ compute the sum and mean as before. Run this version with the same data used in the previous program. You'll see

```
NUMBER 1=?8
SUM=8
NUMBER 2=?3
SUM=11
NUMBER 3=?1
SUM=12
NUMBER 4=?16.5
SUM=28.5
NUMBER 5=?-6.7
SUM=21.8
MEAN=4.36
```

If you're going to input many numbers, you can rewrite this program so that the screen will scroll up every time input is requested. Then the computer will not stop when the screen becomes full.

Going All the Way

Since we're using dimensioned variables, why not go all the way and eliminate SUM? We can use another dimensioned variable, such as M(6), to store the accumulated sum. To use M(6) in place of SUM, change lines 1Ø, 2Ø, 7Ø, 8Ø, and 1ØØ in your program so that it appears as:

```
  5 REM MEAN PROGRAM
 1Ø DIM M(6)
 2Ø LET M(6)=Ø
 3Ø FOR I=1 TO 5
 4Ø PRINT “NUMBER ”;I;“=?”;
 5Ø INPUT M(I)
 6Ø PRINT M(I)
 7Ø LET M(6)=M(6)+M(I)
 8Ø PRINT “SUM=”;M(6)
 9Ø NEXT I
1ØØ PRINT “MEAN=”;M(6)/5
```

Not only have we directly replaced SUM by M(6) in lines 2Ø, 7Ø, 8Ø, and 1ØØ, but we've also had to increase the dimension to M(6), since we need one more dimensioned variable to store the sum.

When you run this program using the same data as before, you'll see no change in the output. Dimensioned variables act just like nondimensioned ones.

Instead of using M(6) to store the sum, we could allow for more dimensioned variables by using any one greater than an index of 5 [e.g., M(<7), M(8), . .] such as

```
1Ø DIM M(2Ø)
```

Defining more dimensioned variables than you need wastes memory, particularly if you only have the standard 2K bytes of memory.

Weird Happenings

Let's see what happens when you begin to use a lot of memory. For the standard 2K computer, change line 1Ø to

```
1Ø DIM M(2ØØ)
```

and run it for the data used before. You'll see the same results. Now enter

```
1Ø DIM M(3ØØ)
```

The first thing you'll notice is that the computer redraws line 1Ø every time you enter a character. Also, the listing and execution appear much slower. Now, as you begin to enter data, everything works fine until you try to input 16.5. The computer won't accept it. Every time you press **ENTER**, the computer displays 16.5. This repeated display indicates that the computer is running out of memory. To continue the program, we first have to free up some memory. One way to do this is to delete remarks. Instead of

```
5 REM MEAN PROGRAM
```

use

```
5 REM MEAN
```

Deleting the word PROGRAM allows you to enter the 16.5, but the computer will stop with a 4/40 report code afterwards. Another way to free up some memory is to erase the contents of the display file—that is, erase the output presently shown on the screen. To erase this output, we must stop the program by holding down **SHIFT** and pressing **DELETE**. Each time you press **DELETE**, you'll back up the L cursor on the line being input. After you delete four times, the L will be at the leftmost position. Now press **SHIFT** and the **A** key to display the keyword **STOP**. Press **ENTER**, and you'll stop with a report code/line number of D/5Ø. Press the **C** key to get a **CONT**, and then press **ENTER** to continue on with the program from the point where you had stopped. Now you can enter the other data, and the program will display the sum and the average, as before.

To see something even more weird, enter

```
        10 DIM M(400)
```

You'll see only lines 5 through 40 displayed, instead of the whole program. In this instance, there's simply not enough room left in memory to display the whole program and keep it in memory.

One way of freeing up some memory now is to delete the values assigned to the variables in the variable area of the program. As we discussed in Volume 1, the program area of memory stores the listing of the program, and the variable area stores the values assigned to the variables during execution. Once the program is completed, we can delete the values in the variable area by using the **CLEAR** command.

Just press the **C** key to print the **CLEAR** command. Now you'll see another weird thing. Instead of the word CLEAR appearing on the bottom line, the first two letters of CLEAR will seem to flutter up the screen. There isn't enough room in memory to show the CLEAR on any line except the one right after line 40. Press **ENTER**, and you will be able to **LIST** the program and see all the lines.

However, there still isn't enough room to run the entire program. Try **RUN**, and you'll just get a report code/line number of 4/10, which means there is not enough room left in memory. With a 16K RAM Module, you're much less likely to run into a problem of insufficient memory capacity.

Although you may see weird happenings on the screen, your program is still safe in memory. You may not be able to run it if available memory becomes slim, but your program won't be lost. You can also save it until you either get more memory or figure out a way to reduce the size of the program. However, you should take note of these weird happenings and start conserving memory, possibly by reusing variables and eliminating REM statements.

Starting from Scratch

Dimensioned variables have one interesting property that regular variables don't have. To show this difference, first reduce the dimension back to the original value of

```
10 DIM M(6)
```

Now delete line 2Ø to eliminate

```
2Ø LET M(6)=Ø
```

If you run the program, you'll see that everything works fine. Why is it that an ordinary variable must first be defined with a **LET**, but a dimensioned variable does not require the same definition? If you had tried to use

```
7Ø LET SUM=SUM+M(I)
```

without a line like

```
2Ø LET SUM=Ø
```

then you would have gotten a report code of 2. The program would stop because the ordinary variable had not first been defined with a separate **LET** statement.

The answer to the question above lies with **DIM**. Besides setting up storage space, the **DIM** also sets the values of all dimensioned variables to zero, as with the following equivalents:

```
LET M(1)=Ø
LET M(2)=Ø
LET M(3)=Ø
LET M(4)=Ø
LET M(5)=Ø
LET M(6)=Ø
```

Since the value of M(6) was initialized, line 2Ø is actually unnecessary. Deleting it saves some memory. So every time a **DIM** is encountered, all the variables are set to Ø.

To show this feature another way, add the line

```
95 DIM M(6)
```

and run the program again. You'll get

```
NUMBER 1=?8
SUM=8
NUMBER 2=?3
SUM=11
NUMBER 3=?1
SUM=12
NUMBER 4=?16.5
SUM=28.5
NUMBER 5=?-6.7
SUM=21.8
MEAN=Ø
```

As you can see, the mean is Ø because all the variables including M(6) were set to zero when the computer encountered **DIM** in line 95.

Variable Dimensions

You can be very economical with dimensioned variables by using exactly the number of variables needed. First, delete line 95 so that you're back to the original program, then change lines to produce the following:

```
 5 REM MEAN
1Ø PRINT "HOW MANY NUMBERS?";
15 INPUT N
2Ø PRINT N
```

```
 25 DIM M(N+1)
 3Ø FOR I=1 TO N
 4Ø PRINT "NUMBER ";I;"=?";
 5Ø INPUT M(I)
 6Ø PRINT M(I)
 7Ø LET M(N+1)=M(N+1)+M(I)
 8Ø PRINT "SUM=";M(N+1)
 9Ø NEXT I
1ØØ PRINT "MEAN=";M(N+1)/N
```

Line 1Ø asks how many numbers to input, and line 15 stores the input as N. Then line 25 dimensions the variables to be one more than the number needed. In other words, if you had input 5, then line 25 would do a **DIM** M(6), just as before. Also, as before, line 7Ø accumulates the sum. If N=5, then line 7Ø becomes effectively

```
7Ø LET M(6)=M(6)+M(I)
```

just as before. Likewise, lines 8Ø and 1ØØ store the results in M(6) and divide M(6) by 5 to get the mean.

Notice how economical this program is. Instead of your having to guess at a large enough value for **DIM** so that any number of items may be handled, this program automatically adjusts the **DIM**.

The speed of execution can also be increased by using a constant for N+1, instead of recalculating it every time we use M(N+1). We could allow the line

```
21 LET D=N+1
```

and then use D each time that N+1 is needed, with the following changes:

```
 25 DIM (D)
 7Ø LET M(D)=M(D)+M(I)
1ØØ PRINT "MEAN=";M(D)/N
```

Another step to consider when using a program is error checking on the input. It's not good to have a program stop with an error message if too much data is input, particularly if the user doesn't know BASIC or hasn't written the program. It is better for a program to operate on a known range of input than to crash unexpectedly. The program should tell the user about conditions it can't handle. To achieve this, we can include the lines

```
22 IF N<2ØØ THEN GOTO 25
23 PRINT "TOO MANY NUMBERS",,,
24 GOTO 1Ø
```

and, thus, not allow 2ØØ or more items to be entered. The actual value could be determined more closely than 2ØØ since we only tried M(2ØØ) and M(3ØØ) in the previous section. Also, lines 21, 22, 23, and 24 take up memory space, too.

What's the Error?

Although it is not really necessary to store the input data just to calculate the mean, storage input is convenient in other calculations. In statistics, one common measure of the spread of data is called the Standard Error. In a large random sample of data, about 68% of the samples fall within one Standard Error, or Standard Deviation of the Mean.

The Standard Error is calculated by the following algorithm:

1. Subtracting the mean from each number that is input
2. Squaring each result
3. Adding up all these squares
4. Dividing by the number of samples, minus one
5. Taking the square root

The purpose in showing this calculation is not to teach statistics, but to illustrate how conveniently your computer can be used with dimensioned variables. Since all the data was stored as it was input, we need only add a few lines of code to the end of our program to determine Standard Error. For our data, the general algorithm is the following:

1. Calculate the mean
2. Subtract the mean from each item of data
3. Square each result
4. Add all the squares
5. Divide by the number of items, minus 1
6. Take the square root of the result

The general formula is as follows:

Standard Error=

$$\sqrt{\frac{(X1\text{-}M)^2+(X2\text{-}M)^2+\ldots+(XN\text{-}M)^2}{N\text{-}1}}$$

For our example,

Standard Error=

$$\sqrt{\frac{(8\text{-}4.36)^2+(3\text{-}4.36)^2+(1\text{-}4.36)^2+(16.5\text{-}4.36)^2+(\text{-}6.7\text{-}4.36)^2}{5-1}}$$

=8.6Ø3662

Now try the following program in which lines 1ØØ to 16Ø have been added to the Mean program.

Notice that in line 12Ø, the variable M(N+1) is set to Ø. This was done so that the variable can be reused to store the sum of the squares. Initially, in line 7Ø, M(N+1) stored the sum of the numbers. Because that sum is no longer needed after line 1ØØ, the M(N+1) can be reused to store the sum of the squares. When you reuse variables, you will have more efficient programs because less memory will be needed to redefine new variables. However, before you reuse a variable, you should make sure you won't need the previous value later on. Also, notice in line 14Ø that we multiply each difference to get the square, which is a more accurate method than using powers such as (M(I)-A)**2.

```
  5 REM MEAN AND STANDARD ERROR
 1Ø PRINT "HOW MANY NUMBERS?"
 15 INPUT N
 2Ø PRINT N
 25 DIM M(N+1)
 3Ø FOR I=1 TO N
 4Ø PRINT "NUMBERS ";I;"=?";
 5Ø INPUT M(I)
 6Ø PRINT M(I)
 7Ø LET M(N+1)=M(N+1)+M(I)
 8Ø PRINT "SUM=";M(N+1)
 9Ø NEXT I
1ØØ LET A=M(N+1)/N
11Ø PRINT "MEAN=";A
12Ø LET M(N+1)=Ø
13Ø FOR I=1 TO N
14Ø LET M(N+1)=M(N+1)+(M(I)-A)*(M(I)-A)
15Ø NEXT I
16Ø PRINT "STANDARD ERROR=";SQR(M(N+1)/(N-
    1))
```

When you run this sequence with the data we've been using, you'll get

```
HOW MANY NUMBERS?
5
NUMBER 1=?8
SUM=8
NUMBER 2=?3
SUM=11
NUMBER 3=?1
```

```
SUM=12
NUMBER 4=?16.5
SUM=28.5
NUMBER 5=?-6.7
SUM=21.8
MEAN=4.36
STANDARD ERROR=8.6Ø3662
```

In a small sample of 5 numbers, the Standard Error is not very reliable, but the concepts developed here can be applied to other problems.

Name Any Month

To provide another example of dimensioned variables, we'll now show how to calculate and print a calendar for any month from 19ØØ to 2Ø99. The following program illustrates many of the BASIC commands we've covered so far:

```
  5 REM CALENDAR MONTH
 1Ø LET COL=4
 2Ø DIM M(12)
 3Ø LET M(1)=31
 4Ø LET M(2)=28
 5Ø LET M(3)=31
 6Ø LET M(4)=3Ø
 7Ø LET M(5)=31
 8Ø LET M(6)=3Ø
 9Ø LET M(7)=31
1ØØ LET M(8)=31
11Ø LET M(9)=3Ø
12Ø LET M(1Ø)=31
13Ø LET M(11)=3Ø
14Ø LET M(12)=31
```

```
15Ø PRINT “NUMBER OF MONTH?”;
16Ø INPUT N
17Ø PRINT N
18Ø PRINT “YEAR?”;
19Ø INPUT Y
2ØØ PRINT Y
21Ø LET L=Ø
22Ø IF Y/4=INT (Y/4) AND NOT Y/1ØØ=INT (Y/1ØØ)
OR Y/4ØØ=INT (Y/4ØØ) THEN LET L=1
23Ø LET DM=M(N)
24Ø IF N=2 AND L THEN LET DM=29
25Ø LET Y=Y-19ØØ
26Ø LET D=365*Y+INT (Y/4)+1
27Ø FOR I=1 TO N-1
28Ø LET D=D+M(I)
29Ø NEXT I
3ØØ IF N<3 AND L THEN LET D=D-1
31Ø LET W=D-7*INT (D/7)
32Ø LET CUR=1
33Ø PRINT “SUN MON TUE WED THU FRI SAT”
34Ø FOR I=Ø TO 5
35Ø FOR J=1 TO 7
36Ø PRINT TAB COL*W;CUR;
37Ø IF CUR=DM THEN STOP
38Ø LET CUR=CUR+1
39Ø IF W=6 THEN GOTO 42Ø
4ØØ LET W=W+1
41Ø NEXT J
42Ø LET W=Ø
43Ø PRINT
44Ø NEXT I
```

Try this program for the current month, and the days will be printed. For example, if the month and year are January, 1983, then you should enter 1 for the month and 1983 for the year, as below:

```
MONTH NUMBER?1
YEAR?1983
SUN MON TUE WED THU FRI SAT
                         1
 2   3   4   5   6   7   8
 9  1Ø  11  12  13  14  15
16  17  18  19  2Ø  21  22
23  24  25  26  27  28  29
3Ø  31
```

In lines 3Ø to 14Ø, the program first initializes the dimensioned variables to the number of days in each month, where month 1 is January and month 12 is December. Line 15Ø next asks for the number of the month for which you want a calendar. You can also change this input to ask that the month be spelled out, and then have the program determine the number of the month.

Line 18Ø asks for which year the calendar month should be calculated. For the algorithm used in this program, years must be greater than 19ØØ. For ease of calculation, line 25Ø subtracts 19ØØ. If you input less than 19ØØ, or a month number less than 1 or greater than 12, you'll get an erroneous answer. If you're going to release a program for general use, you should provide error checking of the input data. For example, you can add the following lines:

```
172 IF N>Ø AND N<13 THEN GOTO 18Ø
174 PRINT “NUMBER NOT VALID. ONLY 1-12”
176 GOTO 15Ø
2Ø2 IF Y>=19ØØ THEN GOTO 21Ø
2Ø4 PRINT “SORRY, YEAR MUST BE >=19ØØ”
2Ø6 GOTO 18Ø
```

Line 21Ø defines a logical variable, L, for convenience. Line 22Ø is the test for a leap year, which was developed in the earlier section on leap years. Line 22Ø sets the variable DM to the number of days in the month. Notice how convenient dimensioned variables are for leap years. Instead of going through 12 **IF** tests, such as

```
IF N=1 THEN DM=31
IF N=2 THEN DM=28
IF N=3 THEN DM=31
. . . . . . . . . . . . . . . . . .
IF N=12 THEN DM=31
```

we simply assign DM=M(N) each time the number of the month is input by the user.

Line 24Ø indicates that if the month is February and the year is a leap year, then there are 29 days in the month. Notice that line 23Ø sets DM=28, if N=2. Therefore, if the test of line 24Ø fails, then February would simply remain at 28.

Line 25Ø subtracts 19ØØ years from the input year for convenience. The year 19ØØ is the base year from which we'll calculate how many days have elapsed to the current date. Actually, lines 25Ø and 26Ø can be combined into the single line

```
LET D=365*(Y-19ØØ)+INT ((Y-19ØØ)/4)+1
```

although it looks more complicated. Lines 26Ø to 3ØØ form the algorithm used in calculating the days elapsed from Sunday, December 31, 1899, to the current date. Since we know that December 31, 1899, was a Sunday, we can calculate the elapsed days, as follows.

Line 26Ø multiplies 365 days times the number of years from 19ØØ. If the year is 1983, then there are 365*83=3Ø295 days. We now have to allow for leap years. Since there is a leap year every four years—except for 19ØØ, 21ØØ, 22ØØ, etc.—we get

```
INT (83/4)=2Ø
```

because there were 20 leap days between 1900 and 1983. So there are 30295+20+1=30316 days. Our algorithm actually works correctly from 1900 to 2099. Since 2100 is not a leap year, we would not include a leap day by the simple calculation of

```
INT (Y/4)
```

To obtain correct information for any year, we must use this general test on every leap year:

```
260 LET D=365*Y+1
262 FOR I=1900 TO Y STEP 4
264 IF Y/4=INT (Y/4) AND NOT Y/100=INT (Y/100)
OR Y/400=INT (Y/400) THEN LET D=D+1
266 NEXT I
```

Again, for the sake of simplifying the program, we've left out the error checking and more exact leap-year calculations.

Lines 270 to 280 simply add up the days from the start of the current year to the current month. Notice here that the upper limit is the number of the month, minus one. For example, if you want January, then N-1=0, and, therefore, line 270 becomes

```
FOR I=1 TO 0
```

In this instance, the body of the loop, line 280, is not executed, and D retains the value it had. If N=2, then we have

```
FOR I=1 TO 1
D=30316+M(1)=30316+31=30347
```

Thus, 30347 days have elapsed from December 31, 1899, to February 1, 1983.

Line 300 checks whether the current month is January or February. If it is either of these, we subtract one leap day that we had calculated.

```
INT (Y/4)
```

If you want to calculate the number of days from one date to another, you must check whether the current date is less than February 28—or greater, since that is the date when the leap day is entered.

Finally, line 31Ø determines what day of the week the first day of the current month falls on. We know how many days, D, have elapsed, and that the weekdays repeat every seven days. The calculation of line 31Ø determines what the day is. If, for example,

```
30347-7*INT (30347/7)=2
```

then the first day of February, 1983, is a Tuesday.

An alternative method of calculating days is to use a formula called Zeller's Congruence. However, the program outlined above, which uses elapsed days, was developed so that you could see the application of many different BASIC statements.

Expanding Dimensions

One way to consider dimensioned variables is in geometric terms. Variables with a single dimension, such as

```
DIM M(1Ø)
```

can be thought of as occupying boxes along a straight line. The concept of dimensioned variables can be extended to two dimensions with the following statement:

```
DIM M(2,3)
```

Figure 1-2 shows the two-dimensional, geometric equivalent of a dimensioned variable. This equivalent is called an *array* of variables because of its geometric representation. In fact, the term array is often used for dimensioned variables, particularly those that are two-dimensioned.

In general, the elements in the **DIM** refer to the number of rows and columns of the array. For Figure 1-2, there are two rows and three columns. Notice that in a row, the first element in parenthe-

M(1,1)	M(1,2)	M(1,3)
1	8.5	−3
0.0015	16	−2.66
M(2,1)	M(2,2)	M(2,3)

Figure 1-2
Geometric representation of a two-dimensional array

ses is constant, but the second element varies with the column number.

Three-dimensional arrays can be compared to an office in a skyscraper. The contents of any office are the people in it. At any time, the number of people in the office may vary, as people leave and enter.

Dimensioned variables with more than three dimensions are hard to think of in terms of geometric analogies.

Two-dimensional arrays are very convenient to use in describing, say, passengers in an airplane. Let's store a number 1 in a variable to represent a filled seat, and a Ø for an empty seat. We'll use small numbers in this example so that you won't have to input many numbers.

```
 5 REM AIRPLANE SEATING
1Ø DIM S(4,2)
2Ø FOR R=1 TO 4
3Ø FOR C=1 TO 2
```

```
40 PRINT "ROW, COLUMN=";R;",";C;
50 PRINT ";SEAT CODE=?";
60 INPUT S(R,C)
70 PRINT S(R,C)
80 NEXT C
90 NEXT R
```

Now enter and run, providing data as follows:

```
ROW,COLUMN=1,1;SEAT CODE=?1
ROW,COLUMN=1,2;SEAT CODE=?1
ROW,COLUMN=2,1;SEAT CODE=?0
ROW,COLUMN=2,2;SEAT CODE=?1
ROW,COLUMN=3,1;SEAT CODE=?0
ROW,COLUMN=3,2;SEAT CODE=?1
ROW,COLUMN=4,1;SEAT CODE=?1
ROW,COLUMN=4,2;SEAT CODE=?1
```

All the data are now stored in the array S.

Searching for a Seat

Since all the data are in S, we can easily add some lines to find out the status of any seat. Add them, but don't run the program.

```
100 PRINT "ROW=?";
110 INPUT R
120 PRINT R
130 PRINT "COLUMN=?";
140 INPUT C
150 PRINT C
160 IF S(R,C)=1 THEN PRINT "SEAT FULL"
170 IF S(R,C)=0 THEN PRINT "SEAT EMPTY"
180 GOTO 100
```

If you run the program, the array S will be cleared of the data you just entered. To save some time, simply do a **GOTO** 1ØØ. This will start the search feature that we just added. Also, if you want to speed up the entering of these lines, then try going to **FAST** mode. If you can get used to the screen flash each time you press a key, you'll see that this mode is much quicker because there is little waiting for a listing every time you enter a line, and the program also is executed much faster.

When the seat search feature is executed, just enter the row and the column numbers as follows:

```
ROW=?1
COLUMN=?1
SEAT FULL
ROW=?2
COLUMN=?1
SEAT EMPTY
ROW=?4
COLUMN=?2
SEAT FULL
ROW=?
```

To stop the program, enter **STOP**. What happens if you try to input a row and a column that has not been dimensioned? As you probably would expect, the program will crash with a report code of 3.

An Ounce of Prevention

To prevent crashes, you must provide error checking of input data. A few lines of code for error prevention can save much time later in trying to solve a problem. For example, add the lines

```
152 IF 1<=R AND R<=4 AND 1<=C AND C<=2
THEN GOTO 16Ø
154 PRINT "SEAT NOT FOUND"
156 GOTO 1ØØ
```

Now try the following as input, and you'll see these results:

```
ROW=?1
COLUMN=?1
SEAT FULL
ROW=?Ø
COLUMN=?9
SEAT NOT FOUND
ROW=?
```

Notice the "SEAT NOT FOUND" message when you try to input row Ø and column 9. It is also possible to be more explicit in the error message by identifying whether it was the row or column that was at fault. This can be done with two separate **IF** tests. One test can be used to check the rows; and the other, the columns.

An **IF** test can also be used to improve the efficiency of this program. The **IF** condition

```
S(R,C)=1
```

is also true if S(R,C) is greater than Ø. Therefore, we can use

```
160 IF S(R,C) THEN PRINT "SEAT FULL"
```

to save a couple of memory bytes. Also, we can use any positive integer for the seat values. Instead of a 1, we can assign a 5 to all filled seats.

Another improvement occurs with the test for empty seats. Instead of a test for

```
S(R,C)=Ø
```

we can use the following:

```
170 IF NOT S(R,C) THEN PRINT "SEAT EMPTY"
```

This test is a byte shorter than the S(R,C)=Ø test. The **NOT** works because it calls for the opposite of a logical operation, as in

```
NOT 1=Ø
```

NOT Ø=1

In fact, **NOT** of any nonzero number is Ø. Therefore, if the seat is empty, then

S(R,C)=Ø

and **NOT** S(R,C)=1. The **IF** condition is true, and the message "SEAT EMPTY" is printed.

CHAPTER 2
Putting Strings in Dimension

Computers were first developed for high-speed calculations, and they do them very well. In fact, the largest computers today can perform over a hundred million additions a second. This kind of speed is making long-range weather forecasting more of a science than an art. Besides performing numerical calculations, computers are being used more and more to process strings of characters. This is very useful in applications like word processing, where the computer acts as an electronic typewriter.

What's in a Name?

Just as dimensioned numeric variables can be applied to help process numerical data, dimensioned string variables can be used with strings. But before we can dimension the strings needed, we have to decide how many characters they will contain.

Dimensioning numeric variables is easy because your computer internally assigns 5 bytes per number. Each number is then precise to about 9 1/2 decimal digits. In most calculations, this degree of precision is sufficient. However, if the computer always assigned 5 bytes, 1Ø bytes, or even 5Ø bytes to every character string, then we could run out of memory storage in the variable storage area. If the character string has 6Ø characters, such as a name and address, you wouldn't want to store only 5Ø characters.

The dimension statement for strings allows you to specify the number of dimensioned strings and how many characters are allocated per string. You must make this decision since the computer can't know in advance how long your strings will be. For example, suppose you want to store 5 names and allocate 1Ø bytes, or characters, per name. You could use this statement

```
10 DIM N$(5,10)
```

in which the first element, 5, in parentheses, is the number of strings, and the second element, 1Ø, indicates that each string has 1Ø bytes of memory reserved for it. Also, the name must be a single alphabetic character followed by a $ sign.

Now let's try the following program to input and print string-dimensioned variables:

```
1Ø DIM N$(5,1Ø)
2Ø FOR I=1 TO 5
3Ø PRINT “NAME ”;I;“=?”;
4Ø INPUT N$(I)
5Ø PRINT N$(I)
6Ø NEXT I
```

Notice that N$(I) refers to the dimensioned variable whose index is I. That is the first index in parentheses. We don't use N$(I,1Ø), just N$(I). Enter and run this program for the names Joe, Dick, Paul, Ann, and Chris, and you'll see the results below:

```
NAME 1=JOE
NAME 2=DICK
NAME 3=PAUL
NAME 4=ANN
NAME 5=CHRIS
```

As you can see, the computer did store the strings in dimensioned variables, then printed the strings of names.

It's in the Stars

But did we get what we asked for? We dimensioned each variable as 1Ø characters long. What data is stored in the memory space we didn't use? To see what's there, change line 5Ø to

```
5Ø PRINT "*";N$(I);"*"
```

in which the asterisks, or stars, will show the beginning and end of the string N$(I). Also, to indicate exactly how many characters are displayed, add the line

```
15 PRINT "Ø123456789Ø123456789"
```

so that you can easily identify columns.

When the program is now run, you'll see

```
Ø123456789Ø123456789
NAME 1=?*JOE       *
NAME 2=?*DICK      *
NAME 3=?*PAUL      *
NAME 4=?*ANN       *
NAME 5=?*CHRIS     *
```

You can see by the stars that every N$(I) does contain 1Ø characters. If characters are not entered for the N$(I), then blanks are stored.

Getting Cut Off

What happens if you input a name with more than 1Ø characters? For our input, let's try

```
ABCDEFGHIJKLMN
```

Now run the program, and you'll see

```
Ø123456789Ø123456789

NAME 1=?*ABCDEFGHIJ*

NAME 2=?
```

As you probably guessed, any characters above 1Ø were truncated, or cut off. This is known as *Procrustean assignment,* a term named after Procrustes, a legendary robber of ancient Greece, who made his victims fit a particular bed by either stretching them if they were too short, or cutting off part of their legs if they hung over the end of the bed. If you try to assign more characters to a dimensioned variable than it is dimensioned for, you'll get Procrustean assignment—the extra characters will be cut off. Only 1Ø characters were stored in N$(I,1Ø) since only 1Ø bytes of storage were allocated for them.

As with nondimensioned strings, you can store any characters in the dimensioned string. Try

```
28Ø1 MAIN
```

and you'll see

```
NAME 2=?*28Ø1 MAIN *

NAME 3=?
```

in which the "28Ø1" is a numeric string—that is, "28Ø1" is composed of numerals, not numbers. The string "28Ø1" is not a number, and it is not stored internally as a single 5-byte number representing the value 28Ø1. The string "28Ø1" is stored as 4 characters.

Packing It In

Is there any way to pack more characters into a dimensioned variable than are allowed by its dimension? Well, yes and no. You can't put in more than 1Ø characters, but there is a way to display more than 1Ø.

You can use BASIC keywords in a way that will enable the computer to print more characters than a variable is dimensioned for. Suppose you want to store the following string:

```
STOP AND THEN RUN SLOW
```

If you try to input this string by spelling out each character, the string will be truncated. But you can use the BASIC keywords to input the string. Just press the keywords on your keyboard instead of spelling out each letter. Try it, and you'll see

```
NAME 3=?* STOP AND THEN RUN SLOW
      *

NAME 4=?
```

in which the last asterisk is printed on the next line because the computer printed past column 31. As mentioned earlier, you can print any printable keyword or function by first using the keyword **THEN** and deleting it later, if desired. Notice that this storage of keywords is very economical, compared to letter-by-letter entry. Only 5 bytes were used to store the 5 keywords (**STOP, AND, THEN, RUN, SLOW**), although 23 letters and spaces were printed on the first line. You can store 1Ø of the BASIC keywords, functions, or other BASIC symbols (e.g., <>, >=, etc.) with a single byte for each one. Internally, the computer stores a special code byte, called a *token,* for each BASIC symbol so that only one byte is needed. This feature needs less memory space for program storage than that required for letter-by-letter storage. The BASIC codes for different symbols are shown in Appendix F. Notice that code 57 represents the letter T, and code 222 represents the BASIC keyword **THEN**.

As with ordinary variables, you can print out the value of dimensioned variables after a program has been run. Try

```
PRINT N$(1),N$(2)
```

and you'll see

```
JOE                         DICK
```

Now try

```
LET N$="JIM"
```

and **PRINT** N$ will show JIM; whereas **PRINT** N$(1), N$(2) will show the same names as before. This result shows that you can have an ordinary string variable with the same name as a dimensioned variable.

How Honest Is Your Computer?

Let's use dimensioned variables to check the honesty of the random number function, **RND**. If **RND** is not really random, then you wouldn't want to use it in computer games, since the results would be biased. Suppose you want to generate random numbers between 1 and 6 to simulate the face of a die as it is thrown. The following program shows how a test of **RND** can be made:

```
10 RAND 1
20 DIM D(6)
30 FOR I=1 TO 100
40 LET R=INT (6*RND+1)
50 LET D(R)=D(R)+1
60 NEXT I
70 FOR I=1 TO 6
80 PRINT I;TAB 5;D(I)
90 NEXT I
```

The **RAND** 1 in line 10 generates all the random numbers from the same point in the pseudorandom sequence. If you delete this line, you will get different answers each time the program is run.

The dimensioned variable, D(6), will store the number of times that a random number is generated. D(1) will store how often a 1 is produced; D(2) will store how often a 2 is produced, etc. Line 30 sets up a **FOR-NEXT** loop to cast the die 100 times. Depending on your patience, you may select a smaller or larger number for the tests.

Line 4Ø generates a random whole number from 1 to 6. Since **RND** returns a number Ø<=**RND**<1, then 6***RND** gives a number that is greater than or equal to Ø and less than 6:

Ø<=6***RND**<6

To get a number greater than or equal to 1, let's add 1 to 6***RND**, as follows:

1<=6***RND**+1<7

To get whole numbers between 1 and 6, we'll simply take the integer function of

6***RND**+1

which gives integers for R so that 1<=R<=6, as shown in line 4Ø.

You might want to write a short test program and observe the numbers being printed to confirm that all of them are in the range from 1 to 6.

Line 5Ø increments the dimensioned variables, which store how often the number has appeared. For example, if a 1 was generated for R and D(1) is initially zero, then line 5Ø gives

```
D(1)=D(1)+1
    =Ø+1
D(1)=1
```

Since dimensioned variables are all initialized to zero, you don't need to use **LET** statements to initialize all the dimensioned variables to zero. Line 6Ø checks if 2ØØØ tests have been done. If not, the computer goes back to line 4Ø and generates another random number.

If the next random number is also a 1, then

```
D(1)=D(1)+1
    =1+1
D(1)=2
```

This shows that a 1 occurred two times.

Lines 7Ø to 9Ø print out how many times each random number is generated.

Try running this program for different numbers of tests. The results are shown below for 1ØØ, 1,ØØØ, and 1Ø,ØØØ tests. Just change line 3Ø to

```
3Ø FOR I=1 TO 1ØØØØ
```

or whatever number of tests you want. For example, in 1ØØ tests, the random number 1 occurs 13 times. In 1Ø,ØØØ tests, the random number 1 occurs 162Ø times. You may also switch to **FAST** mode for the larger tests, to speed up the process.

	Number of Tests		
Random Number	1ØØ	1,ØØØ	1Ø,ØØØ
1	13	163	162Ø
2	2Ø	167	1714
3	22	163	1696
4	12	152	1659
5	18	185	1678
6	15	17Ø	1633

Although a wide variation occurs in only 1ØØ tests, much less variation is evident in 1Ø,ØØØ tries—an indication that **RND** does give an adequate supply of random numbers. If you use the programs for determining Mean and Standard Deviation on these numbers, the results for the 1Ø,ØØØ tests are

```
MEAN 1666.6667

STANDARD DEVIATION=36.329969
```

This display indicates that approximately 68% of our numbers should fall within a spread from 36 above to 36 below 1667; and 95% should fall within two standard deviations from the mean, or from 72 above to 72 below. For our data,

1 Standard Deviation	2 Standard Deviations
1667+36=17Ø3	1667+72=1739
1667-36=1631	1667-72=1595

As you can see, 3, 4, 5, and 6 fall within the range 1631-17Ø3, or within one standard deviation from the mean; and 1 and 2 are within the range 1595-1739, or two standard deviations. For a large number of tests, these deviations indicate that the computer's random numbers approximate true random numbers. Therefore, if you're going to throw dice using the computer, it will be pretty honest over a large number of throws. If you use a different **RAND**, such as **RAND** 2, you will get different numbers; but the final results should be close to those shown here, the Mean and Standard Deviation in particular.

To generate a random number between X and Y, you can use the general formula

INT ((Y-X+1)***RND**+X)

in which Y is a larger number than X. For example, if you want numbers from 1 to 6, then let Y=6 and X=1. This formula gives

INT ((6-1+1)***RND**+1)=**INT** (6***RND**+1)

which is what we just used for the die toss. As another example, try this test program to print 1ØØ random numbers on the screen:

```
 1Ø PRINT "MAX=?";
 2Ø INPUT MAX
 3Ø PRINT MAX
 4Ø PRINT "MIN=?";
 5Ø INPUT MIN
 6Ø PRINT MIN
 7Ø LET F=MAX-MIN+1
 8Ø FOR I=1 TO 1ØØ
 9Ø LET R=INT (F*RND+MIN)
1ØØ SCROLL
11Ø PRINT R
```

```
120 NEXT I
```

The variable F is introduced for convenience. Since MAX-MIN+1 is a constant, there is no need to recalculate it every time in the loop, which would slow down execution speed. Try this program for MAX=1 and MIN=-1, and you'll see

```
 Ø
 Ø
 1
-1
-1
 Ø
 Ø
-1
 Ø
 Ø
-1
 .
 .
 .
 .
```

The Length of It

Your computer has a very useful function, **LEN**, which gives the length of a string. The **LEN** is a shifted function located under the **K** key. For example, enter this program:

```
1Ø PRINT "STRING=?";
2Ø INPUT S$
3Ø PRINT S$
4Ø PRINT "LENGTH=";LEN S$
5Ø GOTO 1Ø
```

and try it for the inputs shown below:

```
STRING=?A
LENGTH=1
STRING=?AB
```

```
LENGTH=2
STRING=?ABC
LENGTH=3
STRING=?12345
LENGTH=5
STRING=?AB12Ø98CD XYZ
LENGTH=13
```

Use the keyword **FAST** and you'll get

```
STRING=? FAST
LENGTH=1
```

Now spell out FAST, letter by letter, and you'll see

```
STRING=?FAST
LENGTH=4
```

Use the symbol on the **T** key for unequal, <>, and you'll get

```
STRING=?<>
LENGTH=1
```

Now, just press the **ENTER** key and you'll see

```
STRING=?
LENGTH=Ø
```

As you can see, the **LEN** function returns the length of a string. Notice that the string "12345" has a length of 5, since it is composed of 5 numerals. Also, you can see in the example underneath "12345" that the space between the D and the X is counted as a character. Next, notice that **FAST** is entered as a keyword, so its length is 1. In the next example, FAST is spelled out, letter by letter, and then its length is 4. Likewise, the <> is input as a keyword, with a length of 1. Finally, in the last example, only the **ENTER** key is pressed. Since no characters are input, the length is Ø. A string with no characters is called an *empty string,* or a *null string.*

When the screen gets full, just press **CONT**. Now try

```
STRING=?“
LENGTH=1
```

If you use the quotation mark on the **P** key, you'll get a syntax error. Since we're inputting a string, we already have quotation marks provided by the computer. So you must use the quote image on the **Q** key. Although the label is “”, what gets printed is just a “, whose length is 1. Now use the quote image to input the string JOE, as

```
“““JOE”””
```

and you'll see

```
STRING=?“JOE”
LENGTH=5
```

because each quotation mark counts as one character.

Guess My Word

Now that you've seen how easy it is to store strings, let's look at an educational application. The following program displays a word for a short time on the screen and then asks you to spell it. You may wish to enter this program in **FAST** mode. Otherwise, you'll have to wait several seconds before you can enter a line when the program gets longer. In fact, **FAST** mode entry becomes a necessity as your programs get longer.

```
 5 REM SPELLING
1Ø PRINT “HOW MANY WORDS?”;
2Ø INPUT N
3Ø PRINT N
4Ø PRINT “MAX CHAR IN A WORD?”;
5Ø INPUT M
6Ø PRINT M
7Ø DIM W$(N,M)
```

```
 8Ø FOR I=1 TO N
 9Ø PRINT "WORD ";I;"=?";
1ØØ INPUT W$(I)
11Ø PRINT W$(I)
12Ø NEXT I
13Ø LET T=1ØØ
14Ø LET R=INT (N*RND+1)
15Ø LET T=T-5
16Ø IF T=-5 THEN LET T=25
17Ø CLS
18Ø PRINT AT 1,1;W$(R)
19Ø FOR I=1 TO T
2ØØ NEXT I
21Ø PRINT AT 1,1;"GUESS MY WORD. USE Ø TO
STOP"
22Ø INPUT A$
23Ø IF A$="Ø" THEN STOP
24Ø FOR I=LEN A$+1 TO M
25Ø LET A$=A$+" "
26Ø NEXT I
27Ø IF A$=W$(R) THEN PRINT AT 1,1;
"RIGHT                      "
28Ø IF A$<>W$(R) THEN PRINT AT 1,1;
"SORRY, WRONG               "
29Ø FOR I=1 TO 5Ø
3ØØ NEXT I
31Ø PRINT AT 1,1;"                           "
32Ø GOTO 14Ø
```

There should be exactly 27 characters in the strings of lines 21Ø, 27Ø, 28Ø, and 31Ø. If not, you'll see some characters left over from a previous string, if it has not been overwritten by a new one. In line 25Ø, there should be one blank between the quotation marks.

This program first stores N words having a maximum of M characters. So, if N=2Ø and M=15, you dimension the words in line 7Ø as

```
W$(2Ø,15)
```

The number of words you can store depends on the memory in your computer. Experiment and see how big you can dimension W$ before you get a report 4 code, which is an out-of-memory message. Even if you do barely manage to get in a large array, the computer will still need some additional memory to run the program and to display it.

Lines 8Ø to 12Ø input and store the words in the dimensioned variables, W$(I). This procedure is the same as the one we followed in the first section of this chapter in storing names. Line 13Ø initializes a variable, T, for timing. This variable is used in the timing loop of lines 19Ø and 2ØØ. Notice this loop literally does nothing. Its only purpose is to prevent line 21Ø from erasing the word by printing over it.

Line 14Ø generates a random number from 1 to N. For example, if you input 2Ø words, then N=2Ø, and

```
1<=R<=2Ø
```

so R can be any integer: 1, 2, 3, 4, 5, 6, 7, 8, 9, 1Ø, 11, 12, 13, 14, 15, 16, 17, 18, 19, or 2Ø.

Lines 14Ø and 32Ø form a loop. Every time the program finishes showing you a word, getting your response, and checking it, the program goes back to line 14Ø to generate another random number. This random number, R, is used to pick a word, W(R), for display.

Line 15Ø subtracts 5 from T to speed up the display. As T gets smaller, the words appear on the screen for briefer periods. You may wish to experiment with the initial value of T, and how it is reduced in line 15Ø. This value of T=1ØØ was chosen so that the words would appear for longer periods at first on the screen, allowing the user to become familiar with the program. In fact, the user may get overconfident, since the display time gets very brief later on when T=5 and T=Ø. Line 16Ø sets T to a moderate display value of 25 if T gets below Ø.

Line 17Ø clears the screen, and then the computer prints the word at 1,1. After the pause in lines 19Ø through 2ØØ, the word is overwritten by the phrase

```
GUESS MY WORD. USE Ø TO STOP
```

Line 23Ø stops the program if you input a Ø.

Lines 24Ø to 26Ø pad the answer with blanks that you input. This padding is necessary because we dimensioned W$ to be of length M. If you input

```
DOG
```

then you'll never get a match with the

```
DOG
```

that was input in lines 8Ø through 12Ø. That's because the DOG from lines 8Ø through 12Ø had M-3 blanks put after it. For example, if M=15, then DOG is stored with 12 blanks after it, as one of the dimensioned variables.

Line 24Ø, then, finds the length of the string you input and pads it with blanks. For example, if you put in DOG, then blanks are added from the fourth to the fifteenth characters, by concatenating blanks to A$ in line 25Ø. The loop of lines 24Ø through 26Ø executes as many times as blanks must be added.

Line 27Ø checks to see if the answer, A$, matches the word, W$(R), which was displayed. If so, the computer prints "RIGHT" from line 27Ø. Line 28Ø checks whether A$ is not equal to W$(R). If A$ is not equal, then "SORRY, WRONG" is printed. Now run this as shown below:

```
HOW MANY WORDS?5
MAX CHAR IN A WORD?5
WORD 1?=DOG
WORD 2?=CAT
WORD 3?=TIGER
WORD 4?=MOUSE
WORD 5?=COW
```

After the last word is entered, the screen will clear, and then you'll see

```
MOUSE
```

appear at the top of the screen for a couple of seconds. This display corresponds to the maximum T=95, since it is reduced by 5 each time a word is displayed.

Next, the computer overwrites MOUSE with

```
GUESS MY WORD. USE Ø TO STOP
```

Notice that this line is 27 characters long. If your words are more than 27 characters, you'll have to add blanks to lines 21Ø, 27Ø, 28Ø, and 31Ø, since the new string must exactly overwrite the previous string that's displayed. Fortunately, there aren't too many words with more than 27 characters. If you enter MOUSE, then the computer comes back with

```
RIGHT
```

for about a second, and then shows you MOUSE again. The second MOUSE occurs because, with only 5 words, there is a good chance that some random numbers will occur several times. To move on, just press **ENTER**, and you'll see

```
SORRY, WRONG
```

appear because the computer thinks you entered 5 blanks. Now the word

```
COW
```

appears and then the prompt

```
GUESS MY WORD. USE Ø TO STOP
```

If you enter Ø, the program stops.

You can save this program with all the words intact simply by a

> **SAVE** "SPELLING"

After you reload the program, do a **GOTO** 13Ø to start displaying words. If you do a **RUN**, then the values of all dimensioned variables are set to Ø, and ordinary variables become undefined.

Being Polite

You should try to program the computer to respond politely to a wrong answer. In a game, you may sometimes want to put in a humorous response like

> "TOO BAD, YOU LANDED ON THE SUN.
> HOPE YOU'VE GOT A GOOD AIR-CONDITIONER."

or

> "YOU BLEW IT. BETTER BRUSH UP ON YOUR
> INTERSTELLAR ASTROGATION. READ 'QUE ON
> INTERSTELLAR TRAVEL' "

In an educational program, however, you should carefully consider the user's feelings. If the user does something wrong and gets a critical reply, such as

> "WRONG, DUMMY! WHY DIDN'T YOU STUDY
> HARDER?"

the user may feel pretty bad and never use the program again! Even a curt response, such as

> "WRONG"

or

> "INCORRECT"

may hurt someone's feelings.

Variations on a Theme

Many variations are possible with this program. For example, you could modify it with additional strings or print statements to show pictures, such as the animals themselves.

Although this program doesn't give a grade, you can add a variable to keep track of the number of correct responses. Instead of the simple **GOTO** loop of lines 140 through 320, you can add a **FOR-NEXT** loop to ask 20 questions with the following statements:

```
135 FOR Q=1 TO 20
320 NEXT Q
```

A more efficient way to modify the program would be to make the total questions asked, Q, some multiple, say 3, of the words N. Then each question would be asked an average of 3 times to reinforce learning. You can make this modification by changing line 135 to

```
135 FOR Q=1 TO 3*N
```

In this way, the user will probably see each word three times.

Another improvement can be made in the random number, R, of line 140. Instead of getting the same word several times in a row, especially if there are not many words, you can check that no two numbers are the same. If they are, then keep executing line 140 until a different value is given to R, one not previously used. For example, add the following lines:

```
125 LET R=1E38
135 LET OLD=R
145 IF R=OLD THEN GOTO 140
320 GOTO 135
```

Line 125 sets the initial value of R to a number that is unlikely to be generated by line 140 for most words. Line 135 stores the value of R in the variable OLD. The new value of R is calculated in line 140, and then line 145 checks to see if the new and old values are equal. If they are equal, the computer goes back to line 140 and generates a new R until the values are not equal.

You can also put a **RAND** statement in

```
135 RAND
```

so no one can know the sequence of random numbers.

Sorting Things Out

Have you ever wished you were more organized? If you have a

book collection
record collection
set of recipe cards
address list
mailing list

or any other items, your computer can help you by sorting them alphabetically. Just input the names or other data, and the computer will maintain them in an alphabetically sorted list. You can use any names; even numbers can be included.

The following program sorts and prints a list. Enter:

```
  5 REM SORT
 10 LET CHAR=20
 20 LET NUM=10
 30 DIM L$(NUM,CHAR)
 40 DIM I$(CHAR)
 50 LET N=0
 60 PRINT "ROOM FOR ";NUM-N;" MORE ITEMS","ITEM?
    USE S TO SAVE, P TO PRINT"
 70 INPUT I$
 80 IF I$(1)="S" AND I$(2)=" " THEN GOTO 260
 90 IF I$(1)="P" AND I$(2)=" " THEN GOTO 210
100 FOR J=1 TO N
```

```
11Ø IF I$<L$(J) THEN GOTO 14Ø
12Ø NEXT J
13Ø GOTO 17Ø
14Ø FOR K=N TO J STEP -1
15Ø LET L$(K+1)=L$(K)
16Ø NEXT K
17Ø LET L$(J)=I$
18Ø PRINT "INSERTED AT POSITION ";J,L$(J)
19Ø LET N=N+1
2ØØ GOTO 6Ø
21Ø CLS
22Ø FOR J=1 TO N
23Ø PRINT J; TAB 3;L$(J)
24Ø NEXT J
25Ø GOTO 6Ø
26Ø SAVE "SORT"
27Ø GOTO 6Ø
```

Line 1Ø sets the variable CHAR to allow 2Ø characters for each item, and line 2Ø sets the variable NUM for 1Ø items. You can redefine CHAR and NUM, depending on what you want to store and how much memory is in your computer. Line 3Ø dimensions L$, the list of dimesioned string variables. Line 4Ø dimensions a string variable, I$, with the same number of characters as the dimensioned string variables, L$. The I$ holds temporarily the input item from line 7Ø. If a nondimensioned variable, I$, were used, it would have to be padded with blanks up to CHAR. If I$ is not the same length as L$, then the sorting will not work because the strings may be of different lengths.

For example, if you store

JACK SMITH

as L$(1), it will be stored with 1Ø trailing blanks after SMITH because L$(1) is dimensioned for 2Ø characters.

Now suppose you enter

JOHN SMITH

and it is stored in an ordinary string variable, I$. Then I$ will have only 1Ø characters in it, and JOHN SMITH will appear to be alphabetically less than JACK SMITH because it has 1Ø characters compared to the 2Ø in JACK SMITH.

Also notice that although I$ is a single dimensioned variable with CHAR characters in line 4Ø, you can write it as I$ instead of I$(1). After you dimension I$(CHAR), your computer will know that you mean the dimensioned variable I$. Assignment to I$ is Procrustean after it has been dimensioned.

If I$ were not dimensioned, you could store a string of any length in it. However, once it is dimensioned, only CHAR characters can be stored in I$; any more than CHAR will not be stored. If the input string is

SMITH,JOHN;1ØØØ MAIN ST, NEW YORK 1ØØ12

then only

SMITH,JOHN;1ØØØ MAIN

will be stored because CHAR=2Ø. By giving a big enough value for CHAR, you can store anything.

In line 5Ø, the variable N is initialized to Ø when the program is first run. N counts the number of items that have been stored. Line 6Ø tells how many more items can be stored, then asks you to

(1) input the name of a new item to be stored, or

(2) enter a P to print out all the stored items, or

(3) enter an S to save the program and all its stored data.

Lines 1ØØ and 12Ø set up a **FOR-NEXT** loop to see if a particular item is on the list. If the item is alphabetically less than the other items, the computer will keep searching. If the item is alphabetically less than L$(J), then the computer has found the place to insert the item.

For example, if the following names are stored:

```
L$(1)="SMITH,JOHN          "
L$(2)="NORTH,TOM           "
L$(3)="WASHINGTON,MARY     "
```

and you input

```
PARKER, JOE
```

the computer will find that I$<L$(3); therefore, it will exit the search loop of lines 1ØØ through 12Ø with J=3.

The loop of lines 14Ø through 16Ø moves the stored item in the list up by one item so that there will be room to insert the new item. For example, assume N=3 and data

```
L$(1)="NORTH,TOM           "
L$(2)="SMITH,JOHN          "
L$(3)="WASHINGTON,MARY     "
```

Notice that there are trailing blanks in each name to make a 2Ø-character total. Suppose "PARKER,JOE" was input as a new name. Then the computer would set

```
L$(3)="SMITH,JOHN          "
L$(4)="WASHINGTON,MARY     "
```

and then assign

```
L$(2)="PARKER,JOE          "
```

from line 16Ø.

Lines 22Ø through 24Ø print out the contents of the list, L$(J). Line 26Ø saves the program and all its variables. When you reload the program, it will automatically start executing from the line following 26Ø where it was saved. Just enter

```
LOAD "SORT"
```

and the program will start up from line 27Ø. If you stop the program and then do a RUN, all the data stored in the dimensioned variable L$ will be erased. In working with programs like this one, you should always have a spare copy of the tape. After the program is saved, you will notice that the last character of the name "SORT" has changed to inverse video. This change is normal when a program is saved in a statement. Just use the regular name SORT to load the program back in.

Now enter and run this program for the maximum of 1Ø items allowed and 2Ø characters per name. Note that no spaces should be inserted after a comma or a semicolon. Try entering

```
SMITH,JOHN;1ØØØ MAIN
NORTH,TOM;3Ø47 JOHNSON
WASHINGTON,MARY;P.O. BOX 98ØØ
```

After you finish entering these names, enter a P to print the stored list. You will see that the items are now arranged alphabetically. Now try entering a 1, then A1, and then A 1. You will see that numbers precede alphabetic characterters. This order occurs because the code of numbers is less than that for alphabetic characters. Notice that the WASHINGTON entry has been reduced to fit into the 2Ø-character limit allowed by the dimension of L$.

Now suppose you want to delete entries from your sorted list. You can delete entries by locating the item to be deleted and moving down the items with higher index numbers. For example, add these lines to delete a stored item:

```
115 IF I$=L$(J) THEN GOTO 28Ø
28Ø FOR K=J TO N-1
29Ø LET L$(K)=L$(K+1)
3ØØ NEXT K
31Ø PRINT "ITEM DELETED",,I$
32Ø LET N=N-1
33Ø GOTO 6Ø
```

Line 115 checks whether I$ is alphabetically equal to each L$(J). If it is, then the computer goes to line 28Ø. Lines 28Ø through 3ØØ effectively delete the entry that is equal to the input by moving all the items in the list down one index number. For example, if J=2 and N=5, then the loop of lines 28Ø through 1ØØ does the following:

```
L$(2)=L$(3)
L$(3)=L$(4)
L$(4)=L$(5)
```

The contents of L$(3), L$(4), and L$(5) are copied into L$(2), L$(3), and L$(4), respectively so that the original contents of L$(2) are lost. Because N=N−1, L$(5)is now the next available storage location.

Try deleting:

```
NORTH,TOM;3Ø47 JOHNSON
```

then do a P to verify that the item was deleted.

You can enhance this program by adding scrolling on inputs and by providing for deletions when just the name is entered, rather than entering the entire item. One way to make this deletion is to store the name as the dimensioned variables

```
DIM N$(NUM,CHAR)
```

and use

```
DIM L$(NUM,CHAR)
```

to store a description of the item. For example, for the 5Øth entry of a record collection, you may have

```
N$(5Ø)="BEATLES            "
L$(5Ø)="A HARD DAYS NIGHT  "
```

The computer would ask

```
"NAME?"
```

and then

"DATA?"

where the input for NAME? is stored in N$, and the input for DATA? is stored in L$.

CHAPTER 3
Strings and Slices

Suppose you have a long string of characters and want to extract part of the string. For instance, in the string

```
A$="JOHN SMITH,1Ø15 MAIN ST."
```

you want to extract the name or the address. Your computer has several string functions that make it easy to insert, extract, and modify strings.

How to Slice a String

An organized group of data is called a *data base.* A common data base is one containing a mailing list. Businesses, clubs, government offices, charity groups, and many others all maintain mailing lists. Even your Christmas card list is a data base. Once you have a data base, you may want to find particular information contained in it.

If a string in your data base contains a name and address, as shown above, you can extract just the name as a *substring*, which is part of a larger string. Your computer has a way of extracting, called *slicing,* which uses the command **TO**. This command is the same symbol as the shifted **4** key. Try the following program:

```
1Ø PRINT "STRING=?";
2Ø INPUT S$
3Ø PRINT S$
4Ø PRINT "START=?";
5Ø INPUT S
6Ø PRINT S
7Ø PRINT "FINISH=?";
8Ø INPUT F
9Ø PRINT F
1ØØ PRINT S$(S TO F)
11Ø GOTO 1Ø
```

Now run the program as follows:

```
STRING=?ABCDE12345
START=?1
FINISH=?1Ø
ABCDE12345
STRING=?ABCDE12345
START=?9
FINISH=?1Ø
45
STRING=?+-12<
START=?2
FINISH=?4
-12
```

In this program, you can extract any part of a string from positions S to F, using line 1ØØ. The expression S$(S **TO** F) extracts, or *slices,* the string S$ so that any part of it can be used. The term to the left of **TO** indicates the starting character position for the slice, and the term after the **TO** indicates the final position for the slice.

These terms are also called subscripts. The first character on the left of the S$ string is at position 1.

What happens if you give a start or a finish number that is not within the boundaries of the string? Let's try it and see.

```
STRING=?AB12
START=?0
FINISH=?4
```

In this case, you'll see a report code error of 3, indicating that the subscript is out of range. The computer uses an internal index to keep track of the characters in a string. The first character is index 1, and the last character is the length of the string. The term *length* indicates the number of characters in the string. If the computer can't match the string index, or subscript, with the starting index of 0 that was input, a report code of 3 is given. You will get the same report code if the finish index is greater than the length of the string. Now suppose we want to extract only names from strings that contain both names and addresses. Try this program:

```
 10 DIM N$(3,30)
 20 LET N$(1)="JOHN SMITH,1001 MAIN ST."
 30 LET N$(2)="MARY JONES,123 E. WASHINGTON"
 40 LET N$(3)="TOM NORTH,P.O. BOX 110"
 50 FOR I=1 TO 3
 60 LET L=LEN N$(I)
 70 FOR J=1 TO L
 80 IF N$(I)(J TO J)="," THEN GOTO 100
 90 NEXT J
100 PRINT "NAME=";N$(I)(1 TO J-1)
110 NEXT I
```

The data in lines 20 to 40 was kept short so that you don't have much typing to do. For a more practical case, you may want to include city, state, zip code, etc. However, the same principles of extracting data will be followed. When you run this program, you will see

```
NAME=JOHN SMITH
NAME=MARY JONES
NAME=TOM NORTH
```

Line 1Ø dimensions 3 strings of 3Ø characters. Lines 5Ø to 11Ø process each dimensioned variable, in turn.

First, N$(1) has its name extracted, then N$(2), and then N$(3). For convenience, line 6Ø defines a variable, L, to hold the length of N$(I). Line 7Ø sets up a **FOR-NEXT** loop to find the position of the first comma in N$(I). The loop of lines 7Ø to 9Ø keeps executing until the comma is found. This comma will be J characters from the left. The loop is exited on line 9Ø, which prints the substring from the first to J-1 characters, representing the name without the comma, which is at position J.

Short Slices

Your computer provides a shorter way of extracting characters from positions J to J. Just use (J), so that line 8Ø becomes

```
8Ø IF N$(I)(J)="," THEN GOTO 1ØØ
```

and you'll get the same results as before.
If you think of a string as being a two-dimensional array, then

```
N$(I,J)
```

is the J'th character in the I'th row of a string array. In an array of numbers, N(I,J) is the J'th number in the I'th row. Figure 3-1 illustrates this concept. It is not possible to identify an individual digit of a number since the decimal number is compressed to a 5-byte binary number. However, it is possible to identify an individual character of a string since each character is stored as a byte. For example,

```
N$(1,1)="J"
N$(1,2)="O"
```

```
N$(1,3)="H"
N$(1,4)="N"
```

You can use

```
N$(I,J)
```

or

```
N$(I)(J)
```

or

```
N$(I)(J TO J)
```

The first notation above is preferred because it requires less memory for storage.

(A) Numeric Array

Row \ Column	1	2	3
1	3	21	−9.76
2	1E13	.0015	74
3	-13.6	66	18.1

(B) String Array

Row \ Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1	J	O	H	N		S	M	I	T	H	,	1	0	0	1		M	A	I	N		S	T	.						
2	M	A	R	Y		J	O	N	E	S	,	9	2	3		E	.		W	A	S	H	I	N	G	T	O	N		
3	T	O	M		N	O	R	T	H	,	P	.	O	.		B	O	X		1	1	0								

Figure 3-1

Now suppose you want to extract the addresses from your data base. In our program, the addresses are in the substring between the first comma and the end of the string. The extraction could be done by adding a line to slice the strings, as follows:

```
1Ø5 PRINT "ADDRESS=";N$(I)(J+1 TO L)
```

Try it, and you will see

```
NAME=JOHN SMITH
ADDRESS=1ØØ1 MAIN ST.
NAME=MARY JONES
ADDRESS=923 E. WASHINGTON
NAME=TOM NORTH
ADDRESS=P.O. BOX 11Ø
```

You can shorten line 1Ø5 to

```
1Ø5 PRINT "ADDRESS=";N$(I)(J+1 TO )
```

and line 1ØØ to

```
1ØØ PRINT "NAME=";N$(I)( TO J-1)
```

Try these new lines, and you'll get the same output as before. The general rules are: (1) if you leave out the term *after* the **TO**, the computer assumes you mean until the end of the string; and (2) if you leave out the term *before* the **TO**, the computer assumes you mean from the first character.

Suppose you leave out both the term before the **TO** and the one after it? Let's try it and see what happens. Change line 1ØØ to

```
1ØØ PRINT "NAME=";N$(I)( TO )
```

and you will get

```
NAME=JOHN SMITH,1ØØ1 MAIN ST.
ADDRESS=1ØØ1 MAIN ST.
NAME=MARY JONES,923 E. WASHINGTON
ADDRESS=923 E. WASHINGTON
```

```
NAME=TOM NORTH,P.O. BOX 110
ADDRESS=P.O. BOX 110
```

The (**TO**) just copies the whole string. It can be abbreviated even further to

```
( )
```

which is the shortest way to write a slice. This notation, however, gives the biggest slice, since it returns the whole string. Notice that a blank line appears after every "NAME=" line that is printed.

When we dimensioned 30 characters in N$(3,30), the blanks in each N$(I) were printed. Since the length of "NAME=" and N$(1), N$(2), and N$(3) exceeds 32 characters, the computer must print the trailing blanks in each N$(I) on a new line.

Slicing has the highest priority of any operation, which means that slicing is always done first. For example, if you want to find the length of the names, you can use the statement

```
107 PRINT "LENGTH OF NAME=";LEN N$(I)( TO J-1)
```

Notice that no parentheses are needed, because the slicing is done before the **LEN** function. Then the **LEN** is applied to the resulting substring.

What's Your Account?

Slicing allows you to insert easily information anywhere in a string. For example, suppose we want to insert a customer account number in our data base of names and addresses. The following program shows the lines that have been added (15, 42, 44, 46, 100, 110, and 120) and the changes:

```
10 DIM N$(3,30)
15 DIM A$(3,6)
20 LET N$(1)="JOHN SMITH,1001 MAIN ST."
30 LET N$(2)="MARY JONES,923 E. WASHINGTON"
40 LET N$(3)="TOM NORTH,P.O. BOX 110"
```

```
 42 LET A$(1)=“1Ø5683”
 44 LET A$(2)=“792641”
 46 LET A$(3)=“347121”
 5Ø FOR I=1 TO 3
 6Ø LET L=LEN N$(I)
 7Ø FOR J=1 TO L
 8Ø IF N$(I,J)=“,” THEN GOTO 1ØØ
 9Ø NEXT J
1ØØ LET N$(I)=N$(I)( TO J)+A$(I)+“,”+N$(I)(J+1 TO )
11Ø PRINT N$(I)
12Ø NEXT I
```

Line 15 supplies three, dimensioned variables—A$(1), A$(2), and A$(3)—that contain the account numbers to be inserted in the N$(1), N$(2), and N$(3). The account numbers are 6 characters long, so we dimension A$(3,6). Lines 42 through 46 contain the account numbers to be inserted.

When the first comma in N$(I) is found in line 8Ø, we exit the loop to line 1ØØ. This concatenates

(1) the portion of N$(I) from the first character up to and including the first comma

(2) the account A$(I) that we want to insert

(3) a comma to end the account number

(4) the rest of N$(I) from just after the comma to the end of N$(I)

When you run this program, you'll see

```
JOHN SMITH,1Ø5683,1ØØ1 MAIN ST.
MARY JONES,792641,923 E. WASHI
TOM NORTH,347121,P.O. BOX 11Ø
```

Although the account numbers were inserted correctly, the “WASHINGTON” was cut off. Can you guess why?

The problem occurs when the account number is added to N$(2), and the resulting string exceeds the 3Ø characters allocated for N$(3,3Ø). Let's now raise the dimension to

```
10 DIM N$(3,4Ø)
```

and run this program again. Now you will see

```
JOHN SMITH,1Ø5683,1ØØ1 MAIN ST.
MARY JONES,792641,923 E. WASHING
TON
TOM NORTH,347121,P.O. BOX 11Ø
```

The complete addresses are printed. However, we have a blank line after the JOHN SMITH and TOM NORTH data because we have 4Ø characters dimensioned for each string. For example, N$(1) contains 31 characters for the name and address of JOHN SMITH, and 4Ø-31 = 9 blank spaces after the address. These blank spaces after the data are called *trailing blanks.* The computer prints the first 32 characters of N$(1) on one line, then has to print 8 trailing blanks on the next line. To get rid of blank lines, you have to avoid printing trailing blanks at the end of a string. One way to avoid them is to start searching from the rear of the string to the first nonblank character encountered, which is the last nonblank character of the string. Count how many blanks were found, slice the string up to this last nonblank character, then print the slice.

The Value of a String

The string value function, **VAL**, gives the arithmetic equivalent of the string. The **VAL** is a shifted function on the **J** key. Enter this test program

```
1Ø PRINT "STRING=?";
2Ø INPUT S$
3Ø PRINT S$
4Ø IF S$="Ø" THEN STOP
5Ø PRINT "VALUE=";VAL S$
```

```
60 GOTO 10
```

and input the keyword symbols for the examples below. Don't spell out the words, or the program will crash. Also, note that in the third example, the function **PI** was used as input, not the letters PI.

```
STRING=?SQR 2
VALUE=1.4142136
STRING=?2+2
VALUE=4
STRING=?PI-3
VALUE=Ø.14159265
```

Now let's try ABC as input. You will see

```
STRING=?ABC
VALUE=
```

and the program will crash at line 5Ø with a report code of 2, because the program believes ABC to be an undefined variable name.

Let's assign a value to ABC. While the program is stopped, enter

```
LET ABC=5
```

then press the **CONT** key to continue where the program stopped.

You'll see

```
VALUE=5
STRING=?
```

printed at the top of the screen. Now the program interprets the string ABC as the name of a defined numeric variable. The computer looks up the value of the variable and prints it out.

Space Saver

The **VAL** can be used to save memory if you need to use the same expression many times. For example, in the monthly payment program in Volume 1, we used the expression

```
INT (1ØØ*N+.5)/1ØØ
```

in a number of lines to round N off to two decimal places. We did manage to conserve some space by defining H=1ØØ and substituting it as

```
INT (H*N+.5)/H
```

An even more efficient way to save space is to use the **VAL** function to define a string, A$, as follows:

```
LET A$="INT (1ØØ*N+.5)/1ØØ"
```

or

```
LET A$="INT (H*N+.5)/H"
```

Then we can use **VAL** A$ to print the value of the string, as shown in the following program:

```
1Ø LET A$="INT (1ØØ*N+.5)/1ØØ"
2Ø PRINT "NUMBER=?";
3Ø INPUT N
4Ø IF N=Ø THEN STOP
5Ø PRINT N
6Ø PRINT "VALUE=";VAL A$
7Ø GOTO 2Ø
```

Line 1Ø defines the string A$ as the expression we want to evaluate. Line 4Ø stops the program when you enter a Ø. Line 6Ø finds the value of the string A$, and line 7Ø just goes back to line 2Ø. Try this program using the following examples:

```
NUMBER=?1ØØ
VALUE=1ØØ
```

```
NUMBER=?1ØØ.123
VALUE=1ØØ.12
NUMBER=?1ØØ.125
VALUE=1ØØ.13
NUMBER=-.ØØ4
VALUE=Ø
NUMBER=1667.8382
VALUE=1667.84
```

As you can see, when the **VAL** function is applied to a string, **VAL** gives the numeric equivalent.

The use of string expressions to replace a function can save a lot of memory if the same expression is used in many lines. The only disadvantage is that the computer takes more time to evaluate

```
VAL A$
```

than

```
INT (1ØØ*N+.5)/1ØØ
```

because the computer must first look up the string A$ and find its value.

To see the time difference, add the lines

```
 7Ø FOR I=1 TO 1ØØØ
 8Ø LET K=VAL A$
 9Ø NEXT I
1ØØ PRINT "DONE"
```

Run the program in **FAST** mode and enter 1ØØ.356 as the number N. You will see that after you press **ENTER** for 1ØØ.356, about 47 seconds elapse before "DONE" is printed. Now change line 8Ø to

```
 8Ø LET K=INT (1ØØ*N+.5)/1ØØ
```

and you'll see that it takes only about 15 seconds before "DONE" is printed.

This difference in time illustrates the trade-off you must make between memory storage and speed. Generally, *statements that increase speed also increase storage.*

There is one restriction that applies to **VAL**. If it is part of an expression, then **VAL** must be written first. Some examples of the right and wrong uses are:

Wrong	Right
LET A=3+**VAL** S$	**LET** A=**VAL** S$+3
PRINT AT VAL A$, **VAL** B$	**LET** B=**VAL** B$ **PRINT AT VAL** A$, B
PLOT VAL A$,**VAL** B$	**LET** B=**VAL** B$ **PLOT VAL** A$, B

Notice that the **PRINT AT** statement must be written as the two separate statements shown in brackets. In a later chapter, we will discuss the **PLOT** and **UNPLOT** commands for plotting on the screen. You must put a **VAL** first for these 2 commands also, and they, too, must be written as at least two statements.

How to Make a String

The opposite of the **VAL** function is the string function, **STR$**, which is a shifted function on the **Y** key. **STR$** converts a number to a string. Enter the program

```
1Ø PRINT "NUMBER=?";
2Ø INPUT N
3Ø IF N=Ø THEN STOP
4Ø PRINT N
5Ø LET S$=STR$ N
6Ø PRINT "STRING=";S$
7Ø GOTO 1Ø
```

and run it for inputs of 1ØØ, -2.67 and 5398.72.

You'll see

```
NUMBER=?1ØØ
STRING=1ØØ
NUMBER=?-2.67
STRING=-2.67
NUMBER=?5398.72
STRING=5398.72
```

Although the string looks just like the number that was input, the string *is* a string, and we can apply the string operators to it. For example, let's add some lines to find the length of the string and locate the decimal point.

```
 7Ø LET L=LEN STR$ N
 8Ø PRINT “LENGTH=”;L
 9Ø FOR I=1 TO L
1ØØ IF S$(I)=“.” THEN GOTO 14Ø
11Ø NEXT I
12Ø PRINT “NO DECIMAL POINT”
13Ø GOTO 1Ø
14Ø PRINT “DECIMAL POINT=”;I
15Ø GOTO 1Ø
```

Lines 9Ø through 11Ø slice up the string S$ until the decimal point is found. If none is found, the computer executes line 12Ø after the loop is done. Try this revised program for the inputs 1ØØ and -2.67. You will see

```
NUMBER=1ØØ
STRING=1ØØ
LENGTH=3
NO DECIMAL POINT
NUMBER=-2.67
STRING=-2.67
LENGTH=5
```

```
DECIMAL POINT=3
```

As expected, 1ØØ has no decimal point, and its length is 3 because it has 3 characters. The length for -2.67 is 5 because it contains 3 digits, plus one decimal point, plus a minus sign. The decimal point is located at the third position from the left, as illustrated here.

Character	Position
-	1
2	2
.	3
6	4
7	5

If you input 5398.72, you will see

```
NUMBER=?5398.72
STRING=5398.72
LENGTH=7
DECIMAL POINT=5
```

Now input 1.234E14, and you'll see

```
NUMBER=?1.234E+14
STRING=1.234E+14
LENGTH=9
DECIMAL POINT=2
```

Notice an interesting thing about this last example. Although you input 1.234E14, the computer printed it with an explicit plus sign after the E, as 1.234E+14. The string stores the input of 1.234E+14. Therefore, there are 9 characters in S$, instead of 8 for 1.234E14.

Secret Codes and Characters

Like all secret codes, the ones that your computer uses are available—if you know where to look. The Appendix of Character

Codes lists the internal codes used by your computer. Notice that some of them, such as

64 through 66

193 through 255

stand for complete symbols, like

197 **VAL**

222 **THEN**

You can see these codes on your screen by using the character function **CHR$**, which is a shifted **U** key. Try

```
PRINT CHR$ 197
```

and you will see **VAL**.

There are 256 possible characters numbered from Ø to 255. Let's use a **FOR-NEXT** loop to view them all. Enter

```
10 FOR I=0 TO 255
20 PRINT CHR$ I;
30 NEXT I
```

Now you can see all the different characters that the computer can print. The characters that the computer can't print are shown as question marks, like the control code **BREAK**. If you'd like to see the nonprinting characters with their corresponding codes, use

```
10 FOR I=0 TO 255
15 SCROLL
20 PRINT I;TAB 5;CHR$ I
30 NEXT I
```

When you run this program, you will see the characters and their codes go scrolling up the screen.

The **CODE** function, which is a shifted **I** key, is the opposite function of **CHR$**.

Enter and run the following program:

```
10 FOR I=0 TO 255
15 SCROLL
20 PRINT I;TAB 5;CHR$ I;TAB 15;CODE CHR$ I
30 NEXT I
```

You will see the same number printed on the left side of the screen as on the right. This happens because the computer prints the number, then the character **CHR$** I, and finally the number whose character is **CHR$** I.

You can use **CHR$** and **CODE** any place a character is used. For example,

```
PRINT "HEL"+CHR$ 49+"O"
```

will print

```
HELLO
```

at the top of your screen since **CHR$** 49 is the same as the letter L. Likewise,

```
PRINT 10+CODE "L"
```

prints a 59 at the top of the screen since **CODE** "L"=49. In fact, if you're really pressed for storage, you can use a **CODE** whose value is the number you want. For example,

```
10 LET A=CODE "L"
```

and

```
10 LET A=49
```

Both assign a value of 49 to A. However, the **CODE** version uses 4 bytes in the program area to store **CODE** "L"; 49 uses 8 bytes. The problem with using **CODE** numbers is that the programs are hard for users to understand.

Security is a Good Code

Many people use a money machine to withdraw money from a bank. You insert your I.D. card in the machine; enter your personal code; then easily, perhaps too easily, withdraw your money. The money machine is a computer that checks with the main bank your request for withdrawal. All of the information needed to verify your withdrawal is transmitted electronically between the money machine and the bank's computer through special cables or the telephone system.

Whatever the communication method, there is always the possibility that someone may try to tap the lines and insert false information. To prevent this, banks and many businesses usually *encrypt* the data being transferred. The term "encrypt" means to convert an ordinary message into a cryptic or secret message. For example, a simple encryption of the message

THIS IS A TEST

might be

UIJT JT B UFTU

If the algorithm for encryption is clever enough, it may be difficult for a criminal to break the code and decipher the message. When the message is received by the appropriate party, it is *decrypted*, or deciphered, back to plain text.

Many different algorithms have been devised to encrypt messages and data. With the popularity of computers and electronic fund transfer (EFT), people have worked hard to produce good encryption algorithms. Many businesses also use encryption to protect information. The new field of computer security is becoming more and more important because so much confidential information is stored in and transmitted between computers.

The string functions of BASIC are convenient for encrypting and decrypting data. Let's see how these functions can perform an encryption with the substitution algorithm, one of the oldest and simplest techniques, in which one character is substituted for another. For the example THIS IS A TEST, the following substitution alphabet was used:

```
   Plaintext ABCDEFGHIJKLMNOPQRSTUVWXYZ
  Ciphertext BCDEFGHIJKLMNOPQRSTUVWXYZA
```

The *plaintext* is the original alphabet, and the *ciphertext* is the enciphered alphabet. For the simple example shown, each letter of the original message is replaced by its successor, and the letter Z is replaced by A. The ciphertext alphabet is called the *key* because it converts plaintext to ciphertext. An *inverse key* converts the encoded message back to plaintext. For the example above, we have

```
 Ciphertext ABCDEFGHIJKLMNOPQRSTUVWXYZ
Inverse key ZABCDEFGHIJKLMNOPQRSTUVWXY
```

In converting from plaintext to ciphertext, you could use the simple approach of having one **IF** statement to convert each letter. If the input plaintext string is stored in the text string T$, then a **FOR-NEXT** loop like the one below could be used. (Note that the line numbers are not shown in this example.)

```
FOR I=1 TO LEN T$
IF T$(I)="A" THEN PRINT "B";
IF T$(I)="B" THEN PRINT "C";
IF T$(I)="C" THEN PRINT "D";
. . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . .
IF T$(I)="Z" THEN PRINT "A";
NEXT I
```

where the dots represent the other **IF** statements used for letters D. . . Y. A similar technique can be used to convert ciphertext back to plaintext.

Instead of using 26 **IF** statements, you can write a much shorter and faster program with string functions, as shown below.

```
 5 REM SUBSTITUTION CODE
1Ø LET C$="BCDEFGHIJKLMNOPQRSTUVWXYZA"
2Ø PRINT "PLAIN TEXT=?"
3Ø INPUT T$
4Ø PRINT T$
5Ø FOR I=1 TO LEN T$
6Ø PRINT C$(CODE T$(I)-CODE "A"+1);
7Ø NEXT I
8Ø PRINT
9Ø GOTO 2Ø
```

In this Substitution program, all of the conversion is done by line 6Ø. This one line uses much less memory and executes faster than the 26 **IF** statements. In the Substitution program, the cipher key is stored in the string variable C$ of line 1Ø.

A run of the Substitution program for plaintext inputs of A, ABC, and the alphabet is shown below. Notice that the correct substitution letters for A, ABC, and the alphabet are listed.

```
PLAIN TEXT=?
A
B
PLAIN TEXT=?
ABC
BCD
PLAIN TEXT=?
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
PLAIN TEXT=?
THIS IS A TEST
UIJT
```

For example, if the plaintext input is T$="A", then line 6Ø first finds the code for "A." If you look up this code in the appendix of

Character Codes, you will see that it is 38. Line 6Ø, therefore, gives

```
PRINT C$(38-38+1)=PRINT C$(1)="B"
```

Similarly, when the input string is ABC, the **FOR-NEXT** loop of lines 5Ø through 7Ø is executed three times, and line 6Ø gives

```
PRINT C$(38-38+1)=PRINT C$(1)="B";
PRINT C$(39-38+1)=PRINT C$(2)="C"
PRINT C$(4Ø-38+1)=PRINT C$(3)="D"
```

and the computer outputs BCD.

A problem arises, however, when you try to input the string

```
THIS IS A TEST
```

The computer stops execution after outputting the ciphertext UIJT with a report code of B/6Ø. As you've probably guessed, no key has been provided for a blank. If you look up the code for a blank in the Appendix of Character Codes, you will see that it corresponds to zero. It would be convenient if the blank code had been defined as 37, because we could then easily modify C$ by including a blank in front of the "A."

Because the code for a blank is Ø, we must include extra statements to substitute for it.

To allow for blanks, we have added to the original Substitution program lines 52, 54, and 56, as shown below. Of course, any other symbols could have been used for the blank key. You can use numbers, punctuation marks, inverse video characters, etc.

```
 5 REM SUBSTITUTION CODE
1Ø LET C$="BCDEFGHIJKLMNOPQRSTUVWXYZA"
2Ø PRINT "PLAIN TEXT=?"
3Ø INPUT T$
4Ø PRINT T$
5Ø FOR I=1 TO LEN T$
```

```
52 IF T$ (I)<>" " THEN GOTO 60
54 PRINT " ";
56 GOTO 70
60 PRINT C$(CODE T$(I)-CODE "A"+1);
70 NEXT I
80 PRINT
90 GOTO 20
```

Some examples of encoding data by substitution and allowing for blanks are shown below.

```
PLAIN TEXT=?
ABCDEFGHIJKLMNOPQRSTUVWXYZ
BCDEFGHIJKLMNOPQRSTUVWXYZA
PLAIN TEXT=?
THIS IS A TEST
UIJT JT B UFTU
PLAIN TEXT=?
```

The substitution alphabet used in this example was chosen for simplicity. You could jumble up the letters in C$ to make your code more difficult to break, as follows:

```
C$="ZAWBCFQSTELNOPDHRIGJKMVUXY"
```

The keyword technique is another encoding technique in which string functions can be used efficiently. First a keyword is chosen. Then the character code for a keyword character is added to the character code of the plaintext character to be encoded. If the character code falls within the range A. . . Z, the character code is printed. (We're assuming here for simplicity that only letters are in the plaintext.) If it does not, then it is adjusted to fall within the range A. . . Z. This process is continued with the next character of the keyword. The following example shows how this encoding works. The character codes have been adjusted so that A=1, B=2, C=3 . . . Z=26 to facilitate explaining the algorithm and the program.

Step Message	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
Character Code for Message	1	2	3	4	5	6	7	8	9	1Ø	11	12	13	14	15	16
(1) Keyword	T	I	M	E	X	T	I	M	E	X	T	I	M	E	X	T
Character Code for Keyword	2Ø	9	13	5	24	2Ø	9	13	5	24	2Ø	9	13	5	24	2Ø
(2) Sum of Keyword and Message Codes	21	11	16	9	29	26	16	21	14	34	31	21	26	19	39	36
(3) If Sum >26 then subtract 26	21	11	16	9	3	26	16	21	14	8	5	21	26	19	13	1Ø
Encoded Message	U	K	P	I	C	Z	P	U	N	H	E	U	Z	S	M	J

The following program uses the keyword algorithm to encode messages.

```
  5 REM KEYBOARD CODE
 1Ø LET C$="TIMEX"
 2Ø PRINT "PLAIN TEXT=?"
 3Ø INPUT T$
 4Ø PRINT T$
 5Ø LET KEYPOINTER=Ø
 6Ø FOR I=1 TO LEN T$
 7Ø IF T$(I)<>" " THEN GOTO 1ØØ
 8Ø PRINT " ";
 9Ø GOTO 15Ø
1ØØ LET KEYPOINTER=KEYPOINTER+1
11Ø IF KEYPOINTER>LEN C$ THEN LET
KEYPOINTER=1
12Ø LET NUMCHAR=CODE C$ (KEYPOINTER)-37+
CODE T$(I)-37
13Ø IF NUMCHAR>26 THEN LET NUMCHAR=
NUMCHAR-26
```

```
14Ø PRINT CHR$ (NUMCHAR+37);
15Ø NEXT I
16Ø PRINT
17Ø GOTO 2Ø
```

The keyword is stored in line 1Ø. The variable KEYPOINTER, defined in line 5Ø, points to the character of the keyword that is currently being used to encode the plaintext. Line 1ØØ increments KEYPOINTER, then line 11Ø tests to see if KEYPOINTER is greater than the length of the keyword, C$. If KEYPOINTER is greater, it is reset to point to the first letter. For the keyword TIMEX, after KEYPOINTER points at the "X," it is reset to point to the first letter, "T," of the keyword, as in Step 1 of the Keyword Example.

Line 12Ø adds the character code and keyword code, as in Step 2 of the Keyword Example. The value 37 is subtracted from each code to reduce it to a value between 1 and 26. For example, if the plaintext character to be encoded was "A," then its character code is 38, and its adjusted value is 38−37=1. Similarly, if the letter is Z, then its adjusted code is 63−37=26. The two −37s could have been combined into a single number −74, but they are separated to help explain their function.

Line 13Ø corresponds to Step 3 of the Keyword Example. Line 14Ø prints the appropriate character. Notice that 37 must be added to NUMCHAR to print a character using the defined codes for the Timex computer. For example, if NUMCHAR=1, then 37 must be added so that the computer will print an "A."

Some examples of plaintext input and the corresponding encoded text are shown below. Notice that each example is encoded correctly except the last one of all A's.

```
PLAIN TEXT=?

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

U K P I C Z P U N H E U Z S M J Z E X R O E J C W T
```

```
PLAIN TEXT=?

AAA

UJN

PLAIN TEXT=?

THE  TIMEX  IS  A  GREAT  COMPUTER

NQR  YGGNK  NQ  U  PEJYN  LBRNOCRW

PLAIN TEXT=?

AAAAAAAAAAAAAAAAAAAAAAAAAAA

UJNFYUJNFYUJNFYUJNFYUJNFYUJ

PLAIN TEXT=?
```

Looking at the output for all A's, you can see a periodic pattern of UJNFY. This pattern is repeated because there are only five characters in the keyword TIMEX. By trying different plaintext input and observing the results, you can get enough information to break any code that uses a keyword.

Many other types of encryption techniques have been studied. In another simple type, transposition, the position of the letters in the plaintext is rearranged. Letters are grouped into blocks and their order is *permuted.* The term "permute" means to rearrange. For example, we could arrange the letters in groups of three and permute them with the pattern 321. For the 321, permute means the first character is put third, the second is left alone, and the third is put first.

Plaintext THE TIMEX IS A GREAT MACHINE

Plaintext grouped in blocks of three characters	**Permuted, 321**
THE	EHT
TI	IT
MEX	XEM

```
  I S          S I
 A   G         G   A
 R E A         A E R
 T   M         M   T
 A C H         H C A
 I N E         E N I
```

The final encoded message is

EHTIT XEMSI G AAERM THCAENI

By itself, the transposition encoding is not difficult to break. If the transposition is combined with substitution, however, it becomes much more difficult to break. For practice, you may want to try some transposition-substitution encoding.

One commonly used commercial cryptographic technique is the Data Encryption Standard (DES). The DES was developed by the U.S. National Bureau of Standards and is based on IBM cryptographic techniques. It has been implemented by both hardware and software techniques. An integrated circuit is commercially available that can encode and decode data very quickly according to the DES standard. The advantage of using hardware is speed. If much data must be encrypted and decrypted, hardware is faster than software. The DES operates by dividing the plaintext message into groups of 64 bits. If each character is represented by 9 bits, then 8 characters are encoded at a time. The 64 bits are then encrypted into 72 quadrillion possible combinations.

Even the fastest computer in the world would take years to break a code encrypted by the DES.

Slicing Months

Slicing can be used to simplify finding the number of days in a month. Instead of using 12 **IF** tests to match the number of a month with its days, as in

```
IF M=1 THEN D=31
```

you can define a string containing all the days, then slice it.

```
1Ø LET A$="3128313Ø313Ø31313Ø313Ø31"
2Ø PRINT "NUMBER OF MONTH=?";
3Ø INPUT N
4Ø PRINT N
5Ø PRINT "DAYS=";A$(2*N-1 TO 2*N)
6Ø GOTO 2Ø
```

The string A$ contains the number of days for each month, from January through December. When you run the program, try 1, 2, and 6:

```
NUMBER OF MONTH=?1
DAYS=31
NUMBER OF MONTH=?2
DAYS=28
NUMBER OF MONTH=?6
DAYS=3Ø
```

The slice is calculated by noting

Month	Character Positions Wanted in A$
1	1,2
2	3,4
3	5,6
4	7,8
5	9,1Ø
6	11,12
7	13,14
8	15,16
9	17,18
1Ø	19,2Ø
11	21,22
12	23,24

For month 1, we want characters 1 and 2; for month 2, we want characters 3 and 4, and so on. The general formula for the first character is

2*N-1

since if N=1

2*1-1=1

if N=2

2*2-1=3

if N=3

2*3-1=5

The formula for the second character is 2*N, so

2*1=2

2*2=4

2*3=6

These formulas were used to compute terms before and after the **TO** in line 5Ø.

Talk about Coincidences!

There is a shorter way of writing the slicing program. Try this version, which has only half as many characters in A$ as before.

```
1Ø LET A$="3Ø3232332323"
2Ø PRINT "NUMBER OF MONTH=?";
3Ø INPUT N
4Ø PRINT N
5Ø PRINT "DAYS=";28+ VAL A$(N)
6Ø GOTO 2Ø
```

When you run this program for any month number, you'll get the number of days in the month. This program works because 28 was subtracted from the number of days in a month. Now we have a single digit number to store in A$, because the string A$ contains the difference between the number of days in the month and 28, as shown below.

Number of Month	A$(N)
1	"3"
2	"Ø"
3	"3"
4	"2"
5	"3"
6	"2"
7	"3"
8	"3"
9	"2"
1Ø	"3"
11	"2"
12	"3"

By taking the **VAL** of A$(N) and adding 28, we get the number of days in the month.

However, there's an even shorter way of finding the number of days in a month because of an amazing coincidence. Try the following version of the previous program, and you'll see that it, too, will give the correct answers. Change line 5Ø to

```
5Ø PRINT "DAYS=";CODE A$(N)
```

which is several bytes shorter than the **VAL** version.

Talk about coincidences! If you look up the codes for the characters Ø through 3, they are

Character	Code
Ø	28
1	29
2	3Ø
3	31

Notice that Ø corresponds to 28, the number of days in February. Likewise, 2 corresponds to 3Ø, and 3 corresponds to 31. Therefore, by just printing the codes for the characters sliced from A$, we get the number of days in the month.

CHAPTER 4
Subroutines

Another name for a program is a *routine*. Part of a program is called a *subroutine*.

Conserving Memory

Subroutines are often used to conserve memory space at the expense of execution speed. When a **GOSUB** statement is used, the computer will go to the subroutine and start executing it. The **GOSUB** is a keyword on the **H** key. A **RETURN** is included in the subroutine to tell the computer to go back to the main program. **RETURN** is the keyword over the **Y** key.

First, enter and run this program, which does not contain subroutines:

```
1Ø PRINT AT 1Ø,1Ø;"HELLO"
2Ø FOR I=1 TO 1Ø
3Ø NEXT I
4Ø PRINT AT 1Ø,1Ø;"          "
5Ø FOR I=1 TO 1Ø
```

```
60 NEXT I
70 GOTO 10
```

Be sure to put 5 spaces in line 40 between the quotation marks, or you won't blank out “HELLO.” This program puts a blinking message on the screen. The **FOR-NEXT** loops of lines 20 to 30 and 50 to 60 control the duration of “HELLO” and the five blanks that erase it. For more interesting effects, try moving the message back and forth or up and down the screen.

In this program, you can see that the same commands are used in lines 20 to 30 and 50 to 60. Instead of repeating the code, we can rewrite this program with a subroutine, as follows:

```
 10 PRINT AT 10,10;“HELLO”
 20 GOSUB 100
 30 PRINT AT 10,10;“     ”
 40 GOSUB 100
 50 GOTO 10
100 FOR I=1 TO 10
110 NEXT I
120 RETURN
```

Since this subroutine is so short, and since we're only going to use it twice, we're not really saving any memory here. This example, however, does illustrate how subroutines work.

When the computer encounters line 20, it goes to line 100 and executes the **FOR-NEXT** loop. This procedure is referred to as a subroutine *call.* Then the computer stores the next line number, 30—which it would otherwise have gone to—in an area of memory called the **GOSUB** *stack* area. Each time a subroutine is called, the line to return to is added to the stack—a process known as *stack push.*

When the loop is done, the **RETURN** in line 120 automatically directs the computer to the line right after the **GOSUB** from which the computer just came. The computer first determines this line number by looking at the last entry in the **GOSUB** stack area.

Then the computer goes to that line number and starts executing from there. Also, the computer automatically deletes this last line number from the stack—a process called a stack *pop.*

The stack is a sequential area of memory. Data is always added and removed from the top end of the stack. The data put in last is the first to be taken out. All of these operations are done automatically, so you don't have to worry about them. The stack provides a convenient way for the computer to keep track of line numbers to return to, even if **GOSUB**'s are nested. In fact, stacks also enable the computer to keep track of nested **FOR-NEXT** loops.

After executing the subroutine called from line 2Ø, the computer returns to line 3Ø, prints the blanks, and does a **GOSUB** to line 1ØØ again. After the loop is over, the computer returns to line 5Ø.

Another version of the program is

```
 1Ø LET A$="HELLO"
 2Ø GOSUB 1ØØ
 3Ø LET A$="          "
 4Ø GOSUB 1ØØ
 5Ø GOTO 1Ø
1ØØ PRINT AT 1Ø,1Ø;A$
11Ø FOR I=1 TO 1Ø
12Ø NEXT I
13Ø RETURN
```

Notice how general this subroutine is now. By defining A$ as the text we want to print, the subroutine will print anything. Another advantage of subroutines is that once they work, they always work (unless a new bug is found). You don't have to write new code every time, if you can use subroutines that have worked before. In addition, when you write a program with subroutines, you can break down a long, complicated program into smaller, more manageable pieces.

An important difference between the **GOSUB** and the **GOTO** is that, with the **GOSUB**, the computer remembers the line it came from. The **RETURN** to the following line is automatic. It would not be as simple to rewrite this program in terms of **GOTO**'s.

Like nested **FOR-NEXT** loops, the **GOSUB**'s can also be nested, meaning that one **GOSUB** can call another **GOSUB**. You can also use variable names or expressions in the **GOSUB**, as in the **GOTO**, to direct the computer. For example, you can use the lines

```
5 LET TIMEDELAY=100

. . . . . . . . . . . . . . . . . .

20 GOSUB TIMEDELAY

. . . . . . . . . . . . . . . . . .

40 GOSUB TIMEDELAY
```

to clarify what the subroutine does. You can also change the value of TIMEDELAY, just as with any variable, and the computer will **GOSUB** the new line number. However, the program may require more memory to use these alterations, depending on how often TIMEDELAY is used.

Putting Them in Their Place

Another point to note about **GOSUB**'s is that they execute faster when they are placed near the beginning of the program. This occurs because the computer always starts looking from the beginning when you give a **GOTO** OR **GOSUB** line number. If you put your **GOTO**'s or **GOSUB**'s further on in your program, the computer will take longer to get to them. There is little noticeable effect on a short program, but the effect is more pronounced with longer programs. To show an example of **GOSUB**'s at the beginning, you could rewrite the program in the previous section as

```
10 PRINT AT 10,10;A$
20 FOR I=1 TO 10
30 NEXT I
40 RETURN
50 LET A$="HELLO"
60 GOSUB 10
```

```
70 LET A$="           "
80 GOSUB 10
90 GOTO 50
```

The disadvantage of placing **GOSUB**'s or **GOTO** destinations at the beginning of your program is that you can't use **RUN** to start the program. If you use **RUN**, execution will always start at the lowest line number, and the computer will give an error message when it encounters the **RETURN** in line 40 without having first executed a **GOSUB**. However, you could add a line to this program

```
1 GOTO 50
```

where 50 is the line number after the **GOSUB**'s.

Subroutines can also call themselves. The following program calculates the factorial of a number using a subroutine that calls itself.

```
  5 REM FACTORIAL
 10 PRINT "NUMBER=?";
 20 INPUT N
 30 PRINT N
 40 LET F=N
 50 GOSUB 100
 60 PRINT "FACTORIAL=?";F
 70 GOTO 10
100 LET N=N-1
110 IF N<=1 THEN RETURN
120 LET F=F*N
130 GOSUB 100
140 RETURN
```

If you run this program for N=1, 2, 3, 4, and 5, you will see

```
NUMBER=?1
```

```
FACTORIAL=1
NUMBER=?2
FACTORIAL=2
NUMBER=?3
FACTORIAL=6
NUMBER=?4
FACTORIAL=24
NUMBER=?5
FACTORIAL=12Ø
```

which are the correct factorials of N.

The subroutine of lines 1ØØ to 14Ø computes the factorial use of a number, where the factorial is written in a general formula for any number, N, as

N!=N*(N-1)*(N-2)*....*2*1

The term "N!" means N factorial. A factorial is the product of the integers from 1 to N. For example, 3! is the product of 1*2*3, or

3!=3*(3-1)*(3-2)=3*2*1

so

3!=6

The important thing about the subroutine calculating factorials is that part of the subroutine is defined in terms of itself. Line 13Ø calls the subroutine again. The value of N is reduced by 1, and then line 11Ø checks if N<=1. If it is, then a **RETURN** is executed back through all the subroutine calls.

A subroutine that is written in terms of itself is called a *recursive* subroutine. It is natural to write the factorial calculation in a recursive way, because the general definition of the factorial is recursive, that is,

N!=N*(N-1)!

Checking Up

Let's look at a practical example of many of the commands we've discussed. At some time or another, everyone has to balance a checkbook. The following program uses the computer as an electronic checkbook, enabling you to store check numbers and amounts. The computer maintains your current balance and allows you to retrieve the data. All the data is saved on tape with the program. When you reload the program, it will automatically run with all the stored data.

When you run the program, it displays a menu of 8 choices.

```
SEE THE CURRENT BALANCE-1
INPUT A NEW BALANCE-2
DELETE CHECKS-3
ADD CHECKS-4
SEE ALL CHECKS-5
SEE CHECKS BY NUMBER-6
VOID CHECKS-7
SAVE THE PROGRAM AND DATA-8
```

Notice that voiding checks restores the amount of the check to the balance. Deleting a check does not restore the check's amount to the balance.

When you press 8 to save, you should already have the recorder running, since the program will then automatically save everything. If you want to delete all the data or start from scratch, just **RUN** the program. After you've filled all the memory space with check data, simply start from scratch with a new tape, and with the final balance as your new balance. Normally, you only want to run the program when you don't want to keep your old records, or when the computer's memory is full. In this way, all your check records can be kept on tape.

```
 5 REM CHECKS
1Ø LET MAX=5
```

```
 2Ø LET CH=1Ø
 3Ø LET Q$=“CHECK NO.?USE Ø TO STOP”
 4Ø LET S$=“2ØØ3ØØ4ØØ6ØØ8ØØ9ØØ97Ø99Ø”
 5Ø LET N=Ø
 6Ø LET B=N
 7Ø DIM C(MAX)
 8Ø DIM A(MAX)
 9Ø DIM I$(MAX,CH)
1ØØ PRINT “SEE BALANCE-1”,,“INPUT NEW
BALANCE-2”,“DELETE-3”,,“ADD-4”,,“SEE
ALL-5”,,“SEE BY NUMBER-6”,,“VOID-7”,,
“SAVE-8”
11Ø INPUT M
12Ø CLS
13Ø GOSUB VAL S$(3*M-2 TO 3*M)
14Ø IF M=8 THEN SAVE “CHECKS”
15Ø GOTO 1ØØ
2ØØ PRINT “BALANCE=$”;B
21Ø RETURN
3ØØ PRINT “BALANCE=?”;
31Ø INPUT B
32Ø PRINT B
33Ø RETURN
4ØØ PRINT Q$
41Ø INPUT M
42Ø IF NOT M THEN RETURN
43Ø GOSUB 2ØØØ
44Ø IF NOT I THEN RETURN
45Ø LET T=A(I)
46Ø FOR J=I TO N-1
47Ø LET C(J)=C(J+1)
48Ø LET A(J)=A(J+1)
```

```
490 LET I$(J)=I$(J+1)
500 NEXT J
510 LET N=N-1
520 GOTO 400
600 PRINT “ROOM FOR ”;MAX-N;“MORE CHECKS”
610 IF NOT MAX-N THEN RETURN
620 PRINT Q$
630 INPUT T
640 IF NOT T THEN RETURN
650 LET N=N+1
660 LET C(N)=T
670 PRINT C(N)
680 PRINT “AMOUNT=?”;
690 INPUT A(N)
700 PRINT A(N)
710 LET B=INT (100*(B-A(N))+.5)/100
720 GOSUB 200
730 PRINT “DATE,PAYEE?”
740 INPUT I$(N)
750 PRINT I$(N)
760 GOTO 600
800 FOR I=1 TO N
810 GOSUB 3000
820 NEXT I
830 RETURN
900 PRINT Q$
910 INPUT M
920 IF NOT M THEN RETURN
930 GOSUB 2000
940 IF NOT I THEN GOTO 900
950 GOSUB 3000
```

```
 96Ø GOTO 9ØØ
 97Ø GOSUB 4ØØ
 98Ø LET B=INT (1ØØ*(B+T)+.5)/1ØØ
 99Ø RETURN
2ØØØ FOR I=1 TO N
2Ø1Ø IF C(I)=M THEN RETURN
2Ø2Ø NEXT I
2Ø3Ø LET I=Ø
2Ø4Ø PRINT "NOT FOUND"
2Ø5Ø RETURN
3ØØØ PRINT "CHECK ";C(I);"      AMOUNT=$";A(I)
3Ø1Ø PRINT I$(I)
3Ø2Ø RETURN
```

Line 1Ø initializes the maximum number of checks allowed, MAX, to 5. If you have the 16K Module, then you can increase the number of checks substantially. However, MAX will also depend on the maximum number of characters, CH, that you allow for both the date and the payee.

Line 3Ø defines variable Q$ as a text string to conserve memory. Instead of storing the string in memory every time, the program will print Q$. Line 4Ø is used to determine the destination of the **GOSUB** in line 13Ø.

Line 5Ø initializes the check number counter, N, to Ø. Each time a check is added, N is increased by 1 in line 65Ø. Each time a check is deleted, N is decreased by 1 in line 51Ø. Line 5Ø initializes the check counter to Ø, because no checks have been entered when the program is first **RUN**. Line 6Ø sets the starting balance, B, to Ø, since N=Ø. The check number is stored in the dimensioned variables C, as set up in line 7Ø.

The amount of each check is stored in the dimensioned variables, A, set up by line 8Ø. The information field, I$, of the check is set up in line 9Ø. Typically, you would store the data and the payee in I$. The value of 1Ø for CH was selected for 1Ø characters of storage in each information variable, I$. This value can be increased to adjust to the amount of memory in your system, as well as your needs. With only a standard 2K T/S 1000, not much memory is available for check storage because of the size of this program.

Line 1ØØ prints out the menu. To minimize the memory storage used, all 8 menu items are stored in one **PRINT** instead of 8 **PRINT**'s. Keywords are used wherever possible to save memory. For example, the original version had a **THEN** before **INPUT**, **NEW**, and **SAVE**. After the desired keyword was entered, the **SHIFT** and the **5** key were used to move the cursor back to the **THEN** and delete it. Likewise, a **THEN** originally preceded the **STOP** in Q$ in line 3Ø. The **THEN** was deleted after the keyword **STOP** was entered. A total of about 2Ø bytes were saved by using keywords instead of spelling out each word.

Line 11Ø inputs your choice of a menu item as the variable M. Then the screen is cleared, and the computer goes to the appropriate subroutine. For example, if M=5, then

S$(3*5-2 **TO** 3*5)=S$(13 **TO** 15)="8ØØ"

and line 13Ø becomes

GOSUB VAL "8ØØ"=**GOSUB** 8ØØ

If you need more subroutines, use 4-digit numbers. For example,

S$="2ØØØ22ØØ24ØØ26ØØ3ØØØ34ØØ44ØØ5ØØØ"

and then

GOSUB VAL S$(4*M-3 **TO** 4*M)

Another alternative is to use the expression

GOSUB 2ØØ*M+1ØØØ

This expression, however, restricts the beginnings of the subroutines to evenly spaced line numbers.

An interesting thing occurs if menu item 8 is input. The computer will do a **GOSUB** to line 99Ø, which is just a **RETURN**. Then the computer will execute line 14Ø, which checks whether item 8 was selected. If it was, then the program and all its data are saved. Line 15Ø will then do a **GOTO** to line 1ØØ where the menu is displayed. If the program is reloaded, it will automatically come up and do a **GOTO** to line 1ØØ, because that is the line after the **SAVE** of line 14Ø. When a **SAVE** is included in a program line and then the program is reloaded, the computer will always start executing at the line following the **SAVE**.

If the computer went to a subroutine such as

```
97Ø SAVE "CHECKS"
98Ø RETURN
```

it would encounter the **RETURN** as its first statement after you reloaded the program, causing a report code error message of 7, and the program would crash.

Interestingly, doing a **SAVE** from a program line also results in the last character of the program name being converted to inverse video. This does not affect the loading, however. Just do a

```
LOAD "CHECKS"
```

or

```
LOAD ""
```

if the tape is positioned properly.

Line 2ØØ displays the balance, B. Lines 3ØØ to 32Ø input and print a new balance, B.

The subroutine starting at line 4ØØ will delete checks. M is the check number to be deleted. Line 42Ø tests whether M=Ø by its logical equivalent. This line is the same as

```
42Ø IF M=Ø THEN RETURN
```

except that a byte of memory is saved by using **NOT** M. If M=Ø, then a **RETURN** is done from line 42Ø back to the main menu. Line 43Ø does a **GOSUB** to 2ØØØ. The subroutine at 2ØØØ finds out if the check number you input matches a stored check. If so, the index, I, of the check number is returned. For example, you might have the following checks stored in

```
C(1)=1000
C(2)=1001
C(3)=1002
. . . . . . . . .
. . . . . . . . .
```

where 1000, 1001, and 1002 are the check numbers, and I is the order in which they were input. The first check stored is I=1, the second is I=2, and so on. If the check is not found, then line 2030 sets I=0, and 2040 prints "NOT FOUND."

Line 440 returns to the main menu if the input check number was not found by seeing whether I=0. Line 450 stores the amount of the check to be deleted in the variable T. This will be used by the subroutine that voids checks instead of just deleting them. The deletion routine cancels checks that have been paid, so the balance is the same after the check is deleted. In the void subroutine of lines 970 to 990, the check is deleted, but its amount is added to the balance by line 980. This amount is added because the check was never paid. Line 980 also rounds off the result to two decimal places to eliminate any error in rounding off.

Lines 460 to 500 perform the deletion of a check by rewriting the later checks over the one to be deleted. For example, suppose

```
C(1)=1000
C(2)=1001
C(3)=1002
N=3
```

Let's delete the first check. For simplicity, we'll show only the deletion of the check numbers. We have

```
FOR J=1 TO 3-1
LET C(J)=C(J+1)
NEXT J
```

Working through this loop, we have

J=1
C(1)=C(2)
J=2
C(2)=C(3)

After this loop is done

C(1)=1ØØ1
C(2)=1ØØ2

and line 51Ø decreases N by 1 to N=2. Line 52Ø goes back to 4ØØ to see if more checks are to be deleted. Lines 6ØØ to 72Ø delete the amount of a check from the balance and store the check information.

Lines 8ØØ to 82Ø print out all stored data about checks. Notice that this subroutine calls another. Lines 9ØØ to 98Ø search for a check by its number and point out all the stored data for this check. Other routines could be added to search for a check by date, payee, etc. Lines 97Ø to 98Ø void a check. Line 99Ø is just a return to preserve the menu technique for **SAVE.**

CHAPTER 5
Graphics and Plotting

One of the best features of the T/S 1000 computer is its ability to plot graphs and figures on the screen. Unlike a simple calculator, the T/S 1000 allows you to create impressive and informative displays.

Computer Graphics and Animation

Although your Timex/Sinclair 1000 is physically small, it has large capabilities. One of its most versatile features is the use of computer graphics for animation. Using graphics characters and loops, you can produce a variety of animation effects.

Enter the following program and run it. You will see a stick figure moving its arms up and down. (The △'s represent blank spaces, and the black squares are graphics symbols on your computer.)

```
 5 REM ANIMATION
10 PRINT AT Ø,Ø;"■△△O△△■"
2Ø PRINT "△■△■△■△"
3Ø PRINT "△△■■■△△"
```

```
 4Ø PRINT “△△△■△△△”
 5Ø PRINT “△△■■■△△”
 6Ø PRINT “△■△△△■△”
 7Ø PRINT “■△△△△△■”
 8Ø FOR I=1 TO 1Ø
 9Ø NEXT I
1ØØ PRINT AT Ø,Ø;“△△△O△△△”
11Ø PRINT “△△△■△△△”
12Ø PRINT “△△■■■△△”
13Ø PRINT “△■△■△■△”
14Ø PRINT “■△■■■△■”
15Ø PRINT “△■△△△■△”
16Ø PRINT “■△△△△△■”
17Ø FOR I=1 TO 1Ø
18Ø NEXT I
19Ø GOTO 1Ø
```

The black squares are inverse blanks. To print a black square, go into graphics mode by pressing **SHIFT** and the **GRAPHICS** (9) key. The L cursor will change to a G indicating that you are in graphics mode. Now, press the **SPACE** key, and a black square will appear. To go back to ordinary printing, press the **SHIFT** and **GRAPHICS** keys again. The G cursor will change back to L. Be sure to put in the spaces between quotation marks exactly as shown; otherwise, the stick figure will be distorted.

Program lines 1Ø through 7Ø print the stick figure with arms up at the top left of the TV screen. Lines 8Ø and 9Ø introduce a small time delay before the next stick figure, from lines 1ØØ through 16Ø, prints over the first figure. This second figure has its arms down. By printing one figure and then another, the illusion of motion appears. The loop of lines 17Ø and 18Ø introduces another time delay, then line 19Ø goes back to print the first stick figure again. The **PRINT AT** statements in lines 1Ø and 1ØØ ensure that the figures are printed on top of each other. If only **PRINT** statements were used, each stick figure would be printed below the preceding one, with no overlap.

To make the stick figure do jumping jacks, change lines 15Ø and 16Ø to

```
15Ø PRINT "△△■△■△△"
16Ø PRINT "△△■△■△△"
```

Now the figure will appear to move both arms and legs.

You can also move objects around the screen. The following program moves a rocket across the screeen. First, the rocket is printed at one location so that you can see its shape. When you enter and run the program, you will see the rocket at the middle left of the screen.

```
 5 REM ROCKET
1Ø LET X=1Ø
2Ø LET Y=Ø
3Ø PRINT AT X,Y;"■■△"
4Ø PRINT AT X+1,Y;"△■■"
5Ø PRINT AT X+2,Y;"■■△"
```

Now, add the following lines to move the rocket from left to right across the screen.

```
6Ø LET Y=Y+1
7Ø GOTO 3Ø
```

As you can see, the rocket does zoom from left to right. However, the black squares from lines 3Ø and 5Ø leave trails. To wipe out the trails, add a blank by changing lines 3Ø, 4Ø, and 5Ø as follows:

```
3Ø PRINT AT X,Y;"△■■△"
4Ø PRINT AT X+1,Y;"△△■■"
5Ø PRINT AT X+2,Y;"△■■△"
```

Notice that an extra blank was also included with line 4Ø so that the three sections of the rocket would stay matched. An alternative form would be

```
3Ø PRINT AT X,Y-1;“△■■△”
5Ø PRINT AT X+2,Y-1;“△■■△”
```

so that line 4Ø would not need to be changed. With this method, however, we can't start at Y=Ø because we would get a negative Y position; so we'd also have to change the starting Y position to

```
1Ø LET Y = 1
```

When you run this version, you will see a single rocket zoom from left to right until the program stops because the rocket has gone past the screen boundary. You will also see the separate sections of the rocket moving because the computer must execute lines 3Ø, 4Ø, and 5Ø in turn.

You can achieve better speed and memory efficiency by combining the three **PRINT AT** statements into one. Delete lines 4Ø and 5Ø and replace line 3Ø with

```
3Ø PRINT AT X,Y;“△■■△”;AT X+1,Y;“△△■■”;AT
X+2,Y;“△■■△”
```

By changing line 6Ø to

```
6Ø LET X=X+1
```

you can move the rocket vertically instead of horizontally. There are two problems, however, that you will encounter when you make this change.

First, the rocket will leave a trail of its body and fins as it moves down the screen. The general solution to this problem is to print blanks over the previous positions to cancel them, which is basically what we did in canceling out the trailing fins of the rocket by adding a blank between each set of quotation marks. In this case, you can add the line

```
4Ø PRINT AT X,Y;“△△△△”
```

where the four blanks between quotation marks cancel out the rocket body.

The second problem is that the rocket body will be traveling sideways against the motion of the rocket. For moving graphics of symmetrical objects, such as an asterisk, a square, or a circle, it doesn't matter what direction of travel they take. But a rocket and other types of objects have a definite front, back and sides. If you want to make the rocket's travel appear natural, then you must redraw it. Try redefining the characters of the rocket for vertical travel.

Plots and Pixels

With the **PRINT AT** command, you can plot over an area of 22 lines by 32 columns for a total of 7Ø4 different positions. Another command of BASIC, called **PLOT**, gives you 4 times the resolution of the **PRINT AT** command; that is, the **PLOT** command allows you to plot four symbols on the screen in the space of a single **PRINT AT** character. However, the **PLOT** only provides black squares, whereas the **PRINT AT** allows you to use any symbols. The **PLOT** is a keyword over the **Q** key.

The general way of writing a **PLOT** command is

```
PLOT X-coordinate, Y-coordinate
```

in which X and Y are the coordinates to be plotted. Figure 5-1 illustrates the plotting grid that covers your screen. You can plot to any one of the cells on this grid, using the **PLOT** command.

Try this program:

```
1Ø PRINT "X=?";
2Ø INPUT X
3Ø IF X=-1 THEN STOP
4Ø PRINT X
5Ø PRINT "Y=?";
6Ø INPUT Y
```

```
70 PRINT Y
80 PLOT X,Y
90 GOTO 10
```

When you run this program for X=Ø, Y=Ø, you will see a small black square printed in the bottom left corner of your screen. This square is the symbol that **PLOT** produces. You can think of using these small squares, called picture elements, or *pixels,* to build anything you want. You can plot up to 22*32*4=2,816 pixels on the screen.

Notice that the second prompt of line 1Ø, "X=?", is printed on the screen immediately following the pixel at Ø,Ø. Now the printing position is set to the position after the last pixel that was plotted. To get around this problem, you should always start printing at the same place on the screen, out of the way of the picture being made. Change lines 1Ø and 5Ø to

```
10 PRINT AT 1,10;"X=?";
50 PRINT AT 2,10;"Y=?";
```

and run the program again for X=Ø, Y=Ø. You will see

```
X=?
```

appear at the top of your screen. Enter a Ø, and a "Y=?" will appear below it. Enter another Ø, and the **PLOT** pixel will appear in the bottom left corner of the screen. Notice that the information at the top of your screen remains

```
X=?Ø
Y=?Ø
```

Now enter a 1, and you will see

```
X=?1
Y=?Ø
```

When you enter another 1, you will see

```
X=?1
```

Y=?1

and the second pixel will be plotted up and to the right of the first. Now enter the following numbers for both X and Y: 2, 3, 4, and 5.

Using this method, you can plot a straight line from the bottom left toward the top right of the screen. If the computer had more resolution—that is, if it could plot smaller pixels—you would see a very thin straight line.

Now enter X=63 and Y=Ø. You will see a pixel appear in the bottom right corner of the screen. This is as far to the right as you can plot, as shown in Figure 5-1. Enter X=63 and Y=43 to get the top right boundary, and X=Ø and Y=43 to get the top left boundary. To plot near the center, use X=32 and Y=22.

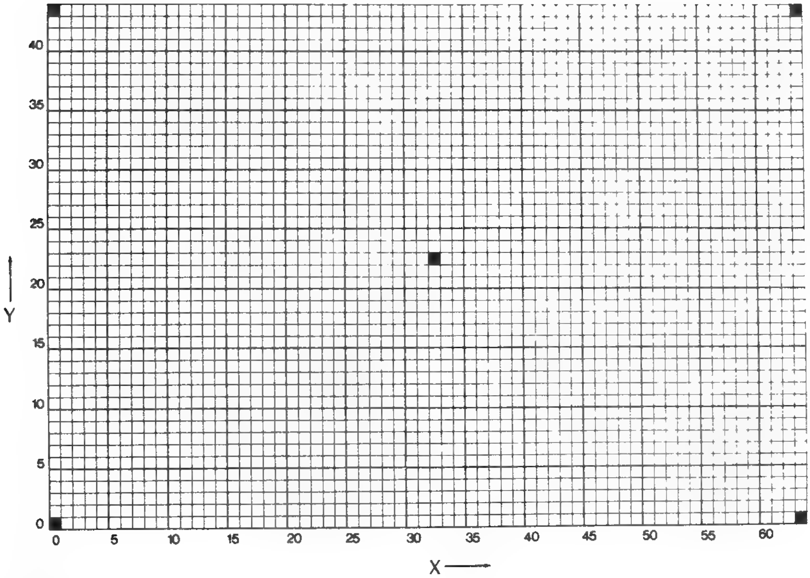


Figure 5-1
Plotting on a grid

If you plot X>63 or Y>43, a report code error of B appears, and execution stops. However, you can plot negative numbers. The **PLOT** command takes the absolute value of whatever it plots. For example, try -1,-2 and you'll see these plot in the same place as 1,2.

Computer Designs

To draw a vertical line, use

```
1Ø FOR I=Ø TO 43
2Ø PLOT Ø,I
3Ø NEXT I
```

and run it.

Now add this line

```
15 PLOT I,I
```

and you'll see a line being plotted at 45 degrees to the vertical line.

Finally, add

```
25 PLOT I,Ø
```

and you'll also see the computer draw a horizontal line. Actually, the horizontal line can go up to 63, but the program would crash with a report code of B when I=44, since the maximum Y value is 43.

You can also plot circles and curves. With them, it is best to plot in the middle of the screen because X- and Y-coordinates can go negative.

```
 5 REM CIRCLE PLOT
1Ø FOR I=Ø TO 2*PI STEP PI/36Ø
2Ø PLOT 2Ø*COS I+32,2Ø*SIN I+22
3Ø NEXT I
```

These lines plot a circle, since the equations of a point moving around the circumference of a circle of radius R is

X=R***COS** I

Y=R***SIN** I

where I is the angle as shown in Figure 5-2.

The constants of 32 and 22 move the circle to the middle of the screen. You may want to run this program in **FAST** mode, because the process takes a while. The **STEP** of **PI**/360 plots in 1/2° increments. For better resolution, such as 1/4° increments, use **PI**/720. However, it takes twice as long to run at 1/4° than at 1/2° increments.

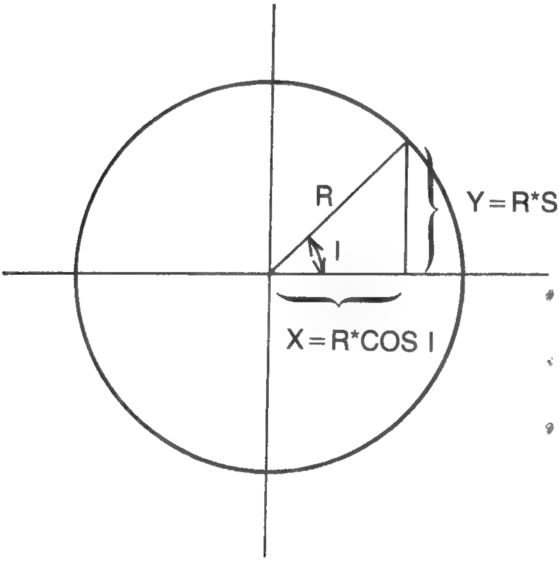


Figure 5-2
Plotting a circle

You can get interesting designs by varying the radius as the circle is plotted. Enter the following program:

```
 5 REM DESIGN PLOT
10 PRINT "NUMBER=?"
20 INPUT N
30 FOR I=Ø TO 2*PI STEP PI/36Ø
40 LET R=2Ø*SIN (N*I)
5Ø PLOT R*COS I+32,R*SIN I+22
6Ø NEXT I
```

and try it for N=4. You'll get a design that looks like a flower with 8 petals. Now try N=Ø.5, and you'll get a sideways heart. You may want to experiment with other values of N or different mathematical functions for R.

Unplotting

The **UNPLOT** command removes a pixel by replacing it with a blank. The **UNPLOT** is a keyword over the **W** key.

The following program demonstrates how unplotting works.

```
 1Ø PRINT AT 1,1Ø;"X=?";
 2Ø INPUT X
 3Ø PRINT X
 4Ø PRINT AT 2,1Ø;"Y=?";
 5Ø INPUT Y
 6Ø PRINT Y
 7Ø PLOT X,Y
 8Ø FOR I=1 TO 1ØØ
 9Ø NEXT I
1ØØ UNPLOT X,Y
11Ø GOTO 1Ø
```

Now run this program where X=Ø and Y=Ø. For a few seconds, the black pixel will remain in the bottom left corner of your screen because of lines 8Ø and 9Ø. Then the **FOR-NEXT** loop of lines 8Ø to 9Ø will finish, and the **UNPLOT** X,Y statement of line 1ØØ will blank out the pixel. The value of 1ØØ in the **FOR** statement of line 8Ø determines how long you will see the pixel. Larger values will keep it there longer.

You can also use the pixel as a cursor on the screen. Depending on a computer terminal's design, the cursor may be blinking or non-blinking. Try the following program to produce a blinking cursor on the screen:

```
 1Ø PRINT AT 1,1Ø;"X=?";
 2Ø INPUT X
 3Ø PRINT X
 4Ø PRINT AT 2,1Ø;"Y=?";
 5Ø INPUT Y
 6Ø PRINT Y
 7Ø PLOT X,Y
 8Ø GOSUB 2ØØ
 9Ø UNPLOT X,Y
1ØØ GOSUB 2ØØ
11Ø GOTO 7Ø
2ØØ FOR I=1 TO 1Ø
21Ø NEXT I
22Ø RETURN
```

Enter X=1, and Y=1. The blinking cursor will appear in the bottom left corner of the screen. To stop this program, you must enter **BREAK**.

Lines 8Ø and 1ØØ introduce a time delay between the plotting and unplotting of the cursor. The program uses a **GOSUB**, because you need the same time delay from **PLOT** to **UNPLOT** as from **UNPLOT** to **PLOT**.

If you remove the **GOSUB**'s from lines 8Ø and 1ØØ, you will see a rapidly blinking cursor. The upper limit of 1Ø in the **FOR** statement of line 2ØØ was chosen to give a blinking period of about 1 second. Try other values for the limit and see how they affect the blinking rate.

Another factor in the blinking rate is the location of the **GOSUB**. In the chapter on Subroutines, you saw that the computer always starts searching for a **GOSUB** or a **GOTO** line number from the beginning of the program. If more lines are added after line 11Ø, the blinking rate will slow down, because the computer must search through more lines until it comes to the **GOSUB**. For the most consistent time delay, put a **FOR-NEXT** loop before the statement you want delayed. You can also put the **GOSUB** body at the beginning of the program, before line 1Ø, to minimize the searching time of the computer. However, this maneuver would require the user to state

```
RUN 1Ø
```

instead of just **RUN** each time through the program. You can get around this restriction if the first line of your program is

```
1 GOTO (Start of Program)
```

In this case, the first line would jump around all the subroutines at the front.

If the **GOSUB** body is put before line 1Ø, you will get an error report code when the computer executes the **RETURN** before line 1Ø without first doing a **GOSUB**.

Moving the Blinking Things

Now that you have a blinking cursor, you can move it using **INKEY$**. Add to the preceding program lines 5 and 72 through 8Ø, as shown below:

```
 5 REM MAZEMAKER
1Ø PRINT AT 1,1Ø;“X=?”;
```

```
 2Ø INPUT X
 3Ø PRINT X
 4Ø PRINT AT 2,1Ø;"Y=?";
 5Ø INPUT Y
 6Ø PRINT Y
 7Ø PLOT X,Y
 72 LET I$=INKEY$
 74 IF I$="8" THEN LET X=X+1
 76 IF I$="5" THEN LET X=X-1
 78 IF I$="6" THEN LET Y=Y-1
 8Ø IF I$="7" THEN LET Y=Y+1
 9Ø UNPLOT X,Y
1ØØ GOSUB 2ØØ
11Ø GOTO 7Ø
2ØØ FOR I=1 TO 1Ø
21Ø NEXT I
22Ø RETURN
```

Line 72 assigns the keyboard input from **INKEY$** to the variable I$. This use of I$ shortens the **IF** tests in the following lines, eliminating the possibility of a change in **INKEY$** during the execution of lines 74 to 8Ø. Such a change could produce an error.

Run this program for X=32 and Y=22, to start the cursor at about the middle of the screen. Press the **8** key, and you will see a line of pixels being drawn to the right. Next, press the **7** key, and the computer will start drawing a vertical line up the screen. Go left with the **5** key and then move down with the **6** key. You can easily draw a box and a big "P." Now reverse your trail, and you will see the cursor moving back along the path. You can make some pretty elaborate mazes and designs with this program.

Unfortunately, you can plot only black pixels with the **PLOT** command. If you want to use the graphics characters or any symbols, you must use **PRINT AT** instead of **PLOT**. For example, make the following changes to your program.

```
7Ø PRINT AT X,Y;"H"
9Ø PRINT AT X,Y;" "
```

In line 7Ø, we will plot an "H." If you want to use an inverse video H, you can perform the following:

(1) Hold down **SHIFT** and press key **9** to get in **GRAPHICS** mode. Notice that the computer's cursor changes from inverse video L to inverse video G, indicating you're in **GRAPHICS** mode.

(2) Press the **H** key, and you can enter an inverse video H after the first quotation mark in line 7Ø.

To get back to ordinary letters, hold down **SHIFT** and press **9** again. The standard L cursor will appear. Next, enter the closing quotation mark.

You can also plot the graphics character on the **H** key. After doing step (1) above, keep holding down **SHIFT** and press **H**. Now, the shifted graphics symbol on **H** will be printed.

If you run your program with X=11 and Y=15, your symbol will start blinking at about the middle of the screen. You can substitute a different graphics symbol, instead of the blank in line 9Ø, and then alternate back and forth between the two symbols. You should notice that the arrow keys don't function in the same way with **PLOT**. As we discussed in an earlier section, **PRINT AT** uses a different coordinate system, with X increasing toward the bottom of the screen and Y increasing toward the right. To make the arrow keys match movement with **PRINT AT**, just change lines 74 through 8Ø. To speed up the drawing, remove lines 1ØØ, 2ØØ, 21Ø, and 22Ø from your program.

A Logical Move

An alternate way of writing lines 74 to 8Ø is with logical expressions. By condensing the 4 lines to 2, you can save memory space and speed up execution. For use with **PLOT** X,Y, replace the 4 lines in 74 to 8Ø with these 2 lines:

```
75 LET X=X+(I$="8")-(I$="5")
80 LET Y=Y+(I$="7")-(I$="6")
```

You will see the same results for the **PLOT** program as before.

Line 75 does a logical check to see if I$="8" and I$="5." If I$="8" is true, then

```
X=X+1+Ø
```

so

```
X=X+1
```

since a 1 is the logical result of a true test, and Ø is the result for a false test.

On the other hand, if I$="5," then

```
  X=X+Ø -1
so X=X-1
```

The same idea is applied to the Y in line 8Ø.

Simple Plots

We can also use the **PLOT** function to draw plots on the screen. The following is a simple plotting program.

```
 5 REM SIMPLE PLOT
1Ø DIM Y(63)
2Ø PRINT "ENTER 1<=X<=63.USE -1
   TO STOP INPUT"
3Ø PRINT "ENTER Ø<=Y<=43"
4Ø LET N=1
5Ø PRINT "X";N;"=";
6Ø INPUT X
7Ø IF X=-1 THEN GOTO 14Ø
8Ø PRINT X
```

```
 9Ø PRINT "Y";N;"=";
1ØØ INPUT Y(X)
11Ø PRINT Y(X)
12Ø LET N=N+1
13Ø GOTO 4Ø
14Ø CLS
15Ø FOR I=Ø TO 63
16Ø PLOT I,Ø
17Ø NEXT I
18Ø FOR I=Ø TO 43
19Ø PLOT Ø, I
2ØØ NEXT I
21Ø FOR X=1 TO 63
22Ø PLOT X,Y(X)
23Ø NEXT X
```

As printed in lines 2Ø and 3Ø of the program above,

```
1<=X<=63
```

and

```
Ø<=Y<=43
```

are the ranges of X and Y, where X is not allowed to be Ø in this simple plot, because the values of Y are stored in the dimensioned variables, Y(X). Unfortunately, the BASIC in the T/S 1000 does not allow Y(Ø).

However, there is a way to get around this restriction. Just define a new variable called

```
Z=X+1
```

and use it in place of X. For example,

```
 95 LET Z=X+1
1ØØ INPUT Y(Z)
```

```
110 PRINT Y(Z)
210 FOR Z=1 TO 63
220 PLOT Z-1, Y(Z)
230 NEXT Z
```

Now, when X=Ø, then Z=1, and so Y(Z)=Y(1), which will work.

This formula illustrates the more general case where X can also be negative. You would find the most negative X value, XN, that was input and then make

```
LET Z=X+ABS XN+1
```

where the absolute value function always returns the positive value.

For example, if the most negative X was -3Ø, then

```
Z=X+3Ø+1
Z=X+31
```

so when X=-3Ø, then Z=1, which is all right.

A more elaborate plotting program would scale the input values to meet the screen boundary conditions of Ø<=X<=63 and Ø<=Y<=43. It would also put tic marks on the X- and Y-axis lines and print scaling numbers along the axis. These are numbers that show the values along an axis, called the *scale*. You may also want to print out a table of the X and Y values to be plotted. Our program is simply designed to illustrate the general concepts of plotting.

Line 4Ø initializes the counter N to Ø. Every time the program requests input, line 4Ø prints the number of the point it is requesting. The Y values are stored in the dimensioned variables Y(X). Now run the simple plot program using the data shown below:

```
ENTER 1<=X<=63.USE -1 TO STOP INPUT
ENTER Ø<=Y<=43
X1=1
```

```
Y1=1
X2=1Ø
Y2=1Ø
X3=2Ø
Y3=2Ø
X4=3Ø
Y4=3Ø
X5=4Ø
Y5=4Ø
X6=-1
```

After the -1 is entered, the computer goes to line 14Ø and clears the screen. Next, it draws a horizontal line to mark the bottom of the plot. This is the X-axis, representing a plot of X when Y=Ø. Line 16Ø,

```
PLOT I,Ø
```

causes the loop variable I to draw the X-axis.

Lines 18Ø to 2ØØ draw the vertical line for the Y-axis. Along the Y-axis, X=Ø, and so we use

```
PLOT Ø,I
```

in line 19Ø to indicate that X=Ø along the Y-axis.

Lines 21Ø to 23Ø plot the stored data. Another enhancement would be to store the number of points to be plotted and only plot that many. For example, if N points were input, then line 21Ø would be

```
21Ø FOR X=1 TO N
```

Putting Up the Bars

In some cases, a histogram, or *bar plot,* is easier to see. In the bar plot, every plotting position up to the value of Y(X) is filled in. Add the following lines to make the bar plot, but don't run it:

```
215 FOR K=1 TO Y(X)
22Ø PLOT X,K
225 NEXT K
```

Instead, do a **GOTO** 14Ø, and you will save yourself the trouble of re-entering all the data. Since the data is stored in the Y(X), you won't lose it by using a **GOTO**, as with a **RUN**.

When you do a **GOTO** 14Ø, you'll see bars rising up to the Y(X). Line 215 plots in every point from 1 to Y(X), using the **PLOT** X,K.You can also plot functions with your computer. Change lines 1Ø to 135 in the bar plot program that follows. The other lines, from 14Ø to 23Ø, remain the same except for line 22Ø.

```
  5 REM FUNCTION PLOT
 1Ø DIM Y(63)
 2Ø PRINT "ENTER 1<=X<=63"
 3Ø PRINT "MINIMUM X=?";
 4Ø INPUT X1
 5Ø PRINT X1
 6Ø PRINT "MAXIMUM X=?";
 7Ø INPUT X2
 8Ø PRINT X2
 9Ø PRINT "FUNCTION=?";
 95 INPUT F$
1ØØ PRINT F$
1Ø5 FOR X=X1 TO X2
11Ø LET Y(X)=VAL F$
115 NEXT X
12Ø LET MAX=Ø
125 FOR X=X1 TO X2
13Ø IF Y(X)>MAX THEN LET MAX=Y(X)
135 NEXT X
138 LET SF=43/MAX
```

```
140 CLS
150 FOR I=Ø TO 63
160 PLOT I,Ø
170 NEXT I
180 FOR I=Ø TO 43
190 PLOT Ø,I
200 NEXT I
210 FOR X=1 TO 63
220 PLOT X,Y(X)*SF
230 NEXT X
```

If you like, you can also make automatic bar plots by adding lines

```
215 FOR K=1 TO Y(X)
220 PLOT X,K*43/MAX
225 NEXT K
```

However, it takes a long time to make bar plots because of the many calculations involved.

If you run the Function Plot program for

```
MINIMUM X=?1
MAXIMUM X=?63
FUNCTION=?X
```

you will see a straight line plotted from the bottom left corner to the top right corner of the screen. It does take a while for all the calculations to be done after you enter the function. Now try

```
MINIMUM=1
MAXIMUM=63
FUNCTION=X*X
```

and you will see a curve plotted.

In the Function Plot program, the values of Y(X) are calculated by lines 1Ø5 to 115. The **VAL** function finds the value of the string for X=X1 to X2.

To keep the plot from running off the screen and crashing the program, a scaling factor is calculated in lines 12Ø to 135. These lines find the maximum Y(X). Initially, MAX=Ø, and the computer starts comparing MAX to every value of Y(X) in the loop of lines 125 through 135. If a Y(X) is greater than MAX, then MAX is assigned the larger value. Lines 14Ø through 23Ø draw the X-axis and the Y-axis as before and plot the Y(X).

Notice how the scaling factor is introduced in line 138 as the maximum Y allowed on the screen, 43, divided by the maximum value found, MAX. In line 22Ø, all the values of Y(X) are multiplied by this scaling factor so that they will stay within the screen boundaries. For example, when Y(X) equals the maximum, then

Y(X)*SF=MAX*43/MAX=43

and the maximum Y(X) is plotted at the maximum screen position of 43.

Try some other functions, such as EXP X, 5*X*X+3*X+4, etc. You can improve this plotting program by scaling for negative values. To do this, you must find the minimum as well as the maximum Y(X). The most general plotting program would allow plotting for positive and negative X- and Y-coordinates.

Going Straight

It is easy to plot straight lines and other geometric shapes with your computer. Figure 5-3 illustrates a straight line passing through X1,Y1 and X,Y. The slope of the line, SL, is, by definition

$$SL = \frac{Y-Y1}{X-X1}$$

which is the same as the tangent of the angle T. If you solve this equation for Y, you get

```
        Y=SL*X+Y1-SL*X1
```

The term Y1-SL*X1 is a constant, which we'll call constant C.

The general formula for a straight line is

```
        Y=SL*X+C
```

where C=Y1-SL*X1.

Also, C is the point on the Y-axis that the line intercepts when X=Ø.

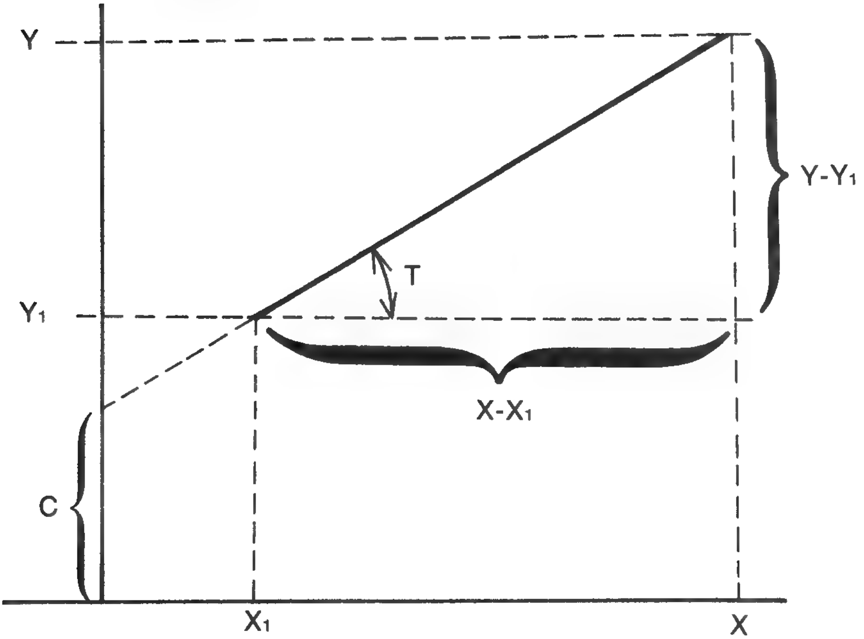
Enter and run this program to draw straight lines:

```
        X1=?Ø
        Y1=?Ø
        X=?63
        Y=?43
```

You will see a straight line plotted from the bottom left corner to the upper right corner of your screen.

```
  5 REM STRAIGHT LINE PLOT
 1Ø PRINT "X1=?";
 2Ø INPUT X1
 3Ø PRINT X1
 4Ø PRINT "Y1=?";
 5Ø INPUT Y1
 6Ø PRINT Y1
 7Ø PRINT "X=?";
 8Ø INPUT X
 9Ø PRINT X
1ØØ PRINT "Y=?";
11Ø INPUT Y
12Ø PRINT Y
13Ø LET SL=(Y-Y1)/(X-X1)
14Ø LET C=Y1-SL*X1
```

```
150 FOR I=X1 TO X
160 PLOT I, SL*I+C
170 NEXT I
```



TAN T=SLOPE $\quad$ TAN T= $\frac{Y\text{-}Y_1}{X\text{-}X_1}$

Figure 5-3
Plotting a straight line

A variable I is used to store the values of X to be plotted.

Now try

```
X1=?Ø
Y1=?43
X=?63
Y=?Ø
```

and a line will be plotted from the upper left to the bottom right of your screen. Notice how the line crashes through the **PRINT** statements at the upper left. To prevent this, use a **CLS** screen command as line 122:

```
122 CLS
```

What happens if you try to plot a vertical line? Try

```
X1=?5
Y1=?Ø
X=?5
Y=?1Ø
```

The program will crash, because the slope is (1Ø-Ø)/(Ø-Ø), and division by zero is not allowed. Vertical lines must be handled as a special case. Before using the slope formula in line 13Ø, we must test if X1=X. If it does, then we can go to the program statements that plot vertical lines:

```
125 IF X<>X1 THEN GOTO 13Ø
126 FOR I=Y1 TO Y
127 PLOT X,I
128 NEXT I
129 STOP
```

Line 125 goes to the regular plotting if X-X1 is not Ø. If X-X1=Ø, then the statements of lines 126 to 128 will plot a vertical line.

Add these lines and run the program again, where

```
X1=?5
Y1=?Ø
X=?5
Y=?1Ø
```

You will see a straight line plotted up.

There are other special cases to consider when you want to plot a straight line. For example, if X1>X or Y1>Y, then you must STEP in a negative direction.

AutoDraw

All of the straight line plots you have done thus far are used in the following program called AutoDraw:

```
  5 REM AUTODRAW
 10 LET P$=“1000”
 20 LET FLAG=0
 30 PRINT “X=?”
 40 INPUT X
 50 PRINT “Y=?”
 60 INPUT Y
 70 CLS
 80 PLOT X,Y
 90 GOSUB 450
100 UNPLOT X,Y
110 LET I$=INKEY$
120 LET X=X+(I$=“8”)-(I$=“5”)
130 LET Y=Y+(I$=“7”)-(I$=“6”)
140 IF I$=“M” AND P$<>“M” THEN GOTO 170
150 LET P$=I$
160 GOTO 80
170 IF FLAG THEN GOTO 220
180 PRINT AT 1,20;“MARK       ”
190 LET X1=X
200 LET Y1=Y
210 GOTO 420
220 PRINT AT 1,20;“NO MARK”
230 LET S=1
```

```
240 IF X<>X1 THEN GOTO 300
250 IF Y1>Y THEN LET S=-1
260 FOR I=Y1 TO Y STEP S
270 PLOT X,I
280 NEXT I
290 GOTO 420
300 IF Y1<>Y THEN GOTO 360
310 IF X1>X THEN LET S=-1
320 FOR I=X1 TO X STEP S
330 PLOT I,Y
340 NEXT I
350 GOTO 420
360 LET SL=(Y-Y1)/(X-X1)
370 LET C=Y1-SL*X1
380 IF X1>X THEN LET S=-1
390 FOR I=X1 TO X STEP S
400 PLOT I,SL*I+C
410 NEXT I
420 LET P$="M"
430 LET FLAG=NOT FLAG
440 GOTO 80
450 RETURN
```

This program automatically draws a straight line between any two points on the screen. When the program is run, it first asks where to put the cursor. You can then move the cursor around the screen by using the keys with arrows. To draw a line between any two points, press the **M** key and the word "MARK" will appear at the top of the screen. This places an invisible mark at the cursor position. Now you can move the cursor to any point and then press **M** again. The words "NO MARK" will appear at the top of the screen, and a straight line will be drawn between the two points.

Slanted lines will look rather ragged, because the pixel size is much larger than a point. Vertical and horizontal lines appear the best. However, this program does give you the chance to draw all kinds of different designs on the screen. To erase a point or a line, move the cursor along it. An erasing cursor is called a *destructive* cursor because it eats up everything in its path.

Line 1Ø initializes a string variable, P$, that contains the previous value of **INKEY$**. Initially, P$ is set to a very large value. In fact, no key press will give such a large value. P$ is used in line 14Ø to tell if your finger is released from one of the cursor movement keys. If it has been released and put down again, the point can be marked. Otherwise, P$ will be set to **INKEY$** in line 15Ø. The variable FLAG in line 2Ø is used to keep track of marked points. If no points have been marked, then FLAG=Ø. If a point has been marked, then FLAG=1. The program tests the value of FLAG in line 17Ø. If FLAG=Ø, then lines 18Ø to 21Ø are executed to print "MARK" on the screen and to store the cursor positions X and Y, in X1 and Y1, respectively.

Lines 8Ø to 16Ø are designed to move the blinking cursor around the screen. Lines 23Ø to 41Ø check the values of X1, X, Y1, and Y to determine how the line should be drawn. The general case is shown in lines 36Ø to 41Ø. Special cases, such as vertical lines, are handled by other sections of the program.

Line 42Ø sets P$="M" after a line has been drawn. No line can be marked until a different value of P$ has been obtained from line 15Ø. Line 43Ø gives the opposite value of FLAG. It is used to avoid having two separate lines in the program, such as

LET FLAG=1

and

LET FLAG=Ø

With line 43Ø, if FLAG=Ø, then **NOT** FLAG=1, and vice versa. Using **NOT** FLAG in line 43Ø takes less code than **LET** FLAG=1, since no number is stored.

CHAPTER 6
Peeking and Poking Around

This chapter is a brief introduction to the advanced commands of BASIC. These commands are very powerful. With them, you can perform many tasks that are impossible with other BASIC commands. For example, in this chapter you will learn how to

(1) find out how much memory your program is using

(2) write a program to renumber lines

(3) create interesting games by detecting collisions between objects on the screen, such as a spaceship and black holes

(4) greatly speed up programs using *machine language,* the only language that the computer really understands

In the Pits

At the machine level, your computer only obeys machine language. This machine language is composed of binary numbers that tell the computer exactly what to do. A single BASIC statement, such as

```
10 LET A=1
```

may be composed of 1Ø machine level instructions.

The microprocessor that runs your computer does not know BASIC. It only knows machine language. The BASIC language is contained in an integrated circuit chip called *read only memory* (ROM). If you'd like to learn more about different types of memory devices and computer hardware, see the books *Foundations of Computer Technology*, and *Modern Computer Concepts,* by J. Giarratano, published by Howard W. Sams.

The microprocessor uses the information in the ROM to interpret each BASIC command or statement into machine language. Then the microprocessor executes those machine-language instructions and interprets the next BASIC statement. The version of BASIC used in the T/S 1000 is called an *interpreter,* because the microprocessor interprets each line of BASIC into machine code as it is executed. Other versions of BASIC use a program called a *compiler* to translate all the BASIC statements into machine code. Instead of interpreting only one line at a time and then executing it, a compiled BASIC only has to execute the machine-language instructions. All the BASIC statements have been translated into machine language before execution starts. A compiled BASIC program operates 1Ø or more times faster than interpreted BASIC. However, you must wait until the program is compiled before it can execute. Also, it is harder to debug a compiled language, because it is difficult to tell which machine-language instructions correspond to the BASIC statements.

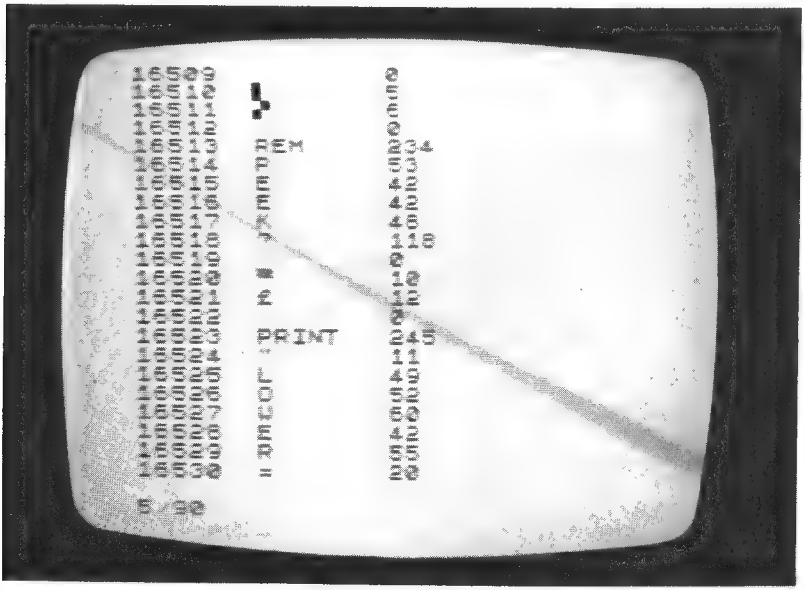
Peeking Around

The **PEEK** command allows you to look into any memory location of the computer. **PEEK** is a shifted function under the **O** key. You can see the contents of any byte of memory by using this function. For example, enter the program below. Be sure to enter the statements exactly as shown, and spell out PEEK in line 5; otherwise you will be peeking at different addresses than those described here if you use the token **PEEK**.

```
5 REM PEEK
```

```
 10 PRINT "LOWER=?";
 20 INPUT L
 30 PRINT L
 40 PRINT "UPPER=?";
 50 INPUT U
 60 PRINT U
 70 FOR I=L TO U
 80 LET P=PEEK I
 90 PRINT I;TAB 7;CHR$ P;TAB 15;P
100 NEXT I
```

Run this program for L=16509 and U=17000, and you will see the first screenful of output shown in Figure 6-1. The number at the left is the memory address. Following it is the character, **CHR$**, corresponding to the peeked value P. Also shown is the peeked value itself, starting in column 15.



```
16509          0
16510  ▌       5
16511  ▚       6
16512          0
16513  REM     234
16514  P       53
16515  E       42
16516  E       42
16517  K       48
16518  ?       118
16519          0
16520  ■       10
16521  £       12
16522          0
16523  PRINT   245
16524  "       11
16525  L       49
16526  O       52
16527  W       60
16528  E       42
16529  R       55
16530  =       20

5/90
```

Figure 6-1
PEEK program output

The starting address, 165Ø9, is always the beginning of the program memory. The contents of address 165Ø9 and address 1651Ø represent the line number. Each memory address can store one byte. Therefore,each address can store any number from Ø to 255, as shown below:

Binary	Decimal
ØØØØØØØØ	Ø
ØØØØØØØ1	1
ØØØØØØ1Ø	2
ØØØØØØ11	3
ØØØØØ1ØØ	4
....	.
....	.
11111111	255

A binary number is composed of *binary digits,* called *bits.* Thus, a byte can contain 2^8=256 numbers from Ø to 255.

Theoretically, the two bytes that start each line of a BASIC statement can hold line numbers from Ø to 65535. However, the BASIC used in the T/S 1000 is designed, normally, to accept line numbers from 1 to 9999 only.

Line numbers of the first line up to 256 are stored in address 1651Ø. If the line number is 256, then a 1 is put in address 165Ø9 and a Ø in 1651Ø. This occurs because

1*256+Ø=256

The first address of 165Ø9 contains the *high byte,* and the second address of 1651Ø contains the *low byte.* The high byte is the integer part of the line number/256. The low byte is the remainder after the line number is divided by 256, or

Line number—256***INT**(line number/256)

Some examples are shown as follows:

High Byte	Low Byte	Line Number
ØØØØØØØØ	ØØØØØØØ1	1
ØØØØØØØØ	ØØØØØØ1Ø	2
ØØØØØØØØ	ØØØØØØ11	3
ØØØØØØØ1	ØØØØØØ11	259
ØØØØØØ1Ø	ØØØØØØ11	515

The last example is

2*256+3=515

Notice that the characters corresponding to the line numbers of 165Ø9 and 1651Ø have no meaning. The characters printed by **CHR$** have meaning only with the BASIC commands.

The next 2 bytes, from addresses 16511 and 16512, contain the length, in bytes, up to the end of the line. If you look at address 16518, you will see that it contains a 118. Referring to the Appendix of Character Codes, you can see that a code of 118 is an **ENTER**. This **ENTER** is the end of the line for every BASIC statement. Starting with the **ENTER**, you can count back 6 memory addresses to the **REM** in address 16513, which was the first command of the line. If you look above it, to address 16511, you will see a 6. Thus, addresses 16511 and 16512 store the number of bytes in the first statement. This length does not include the 2 bytes of the line number or the 2 bytes for the line's length, but the length does include the **ENTER** code 118.

The 2 bytes following the line number contain the length of the line. The order of their storage is opposite to the way line numbers are stored. Instead of high byte, low byte storage, the length is stored as low byte and then high byte. For example, if there were 256 bytes in the first line of the program, then

Address	Contents
16511	Ø
16512	1

because

```
1*256+Ø=256
```

All the numbers in the T/S 1000 are stored as low byte, high byte, with the exception of the line numbers.

Addresses 16513 through 16517 contain the code for each character of the BASIC statements. Notice that the code 234 in address 16513 is enough for the computer to recognize this as a **REM** token. You can check this yourself by looking for code 234 in the Appendix on Character Codes.

If you look down at addresses 16519 and 1652Ø, you will see that they contain a Ø and a 1Ø. Therefore, the next line number is

```
Ø*256+1Ø=1Ø
```

Addresses 16521 and 16522 contain a 12 and a Ø, so there are 12 bytes in line 1Ø, where the bytes for the line number and its length are not included. Press **CONT** and you will see the next page on the screen. Line 1Ø ends with the 118 at address 16534. Counting back to address 16523 shows that there are indeed 12 bytes in the line.

Let's see how numbers are stored. Add the following line and run the program again.

```
1 LET A=1Ø
```

Now address 165Ø9 contains a Ø, and 1651Ø contains a 1. Next, you will see 12 bytes in the line, followed by the token of 241 for **LET**. Then 241 is followed by the token of 38, for A, at address 16514, the = sign, and the number 1Ø. The 1Ø is stored as 29,28. Following it is

```
126
132
 32
  Ø
  Ø
```

```
        Ø
      118
```

The 118 marks the end of the line, and the five numbers above it are used to represent 1Ø in binary form for the computer. The 126 is a special character put in to mark the beginning of a number's internal form. The numbers 132, 32,Ø,Ø,Ø represent the 1Ø as 5 bytes to the computer. The way the computer actually stores numbers is more complicated than a simple binary representation. The storage method is called a *normalized radix point representation.* (For more details, see *BASIC: Advanced Concepts,* by J. Giarratano, published by Howard W. Sams, Inc.) If you keep peeking, you will finally see the **NEXT** I and the last 118 at address 16653. It helps to do all this peeking in **FAST** mode and then press **CONT** to see the next screenful of data.

The 118 at address 16654 indicates the end of the program and the beginning of the display file. The display file always begins with an **ENTER**, which the computer puts in automatically. The following numbers correspond to the addresses and data of the display file. Notice that the characters shown here match the first address at the top of your screen:

```
16654     ?
16655     1
16656     6
16657     6
16658     3
16659     9
```

The starting addresses of the display file float above the program area. As the program increases, so does the starting address of the display file. Its starting address is kept in the system variables area of 16384 through 165Ø8. The display file address is kept in addresses 16396 and 16397. Try this

```
PRINT PEEK 16396+256*PEEK 16397
```

and you will get 16654, the start of the display file if you have the

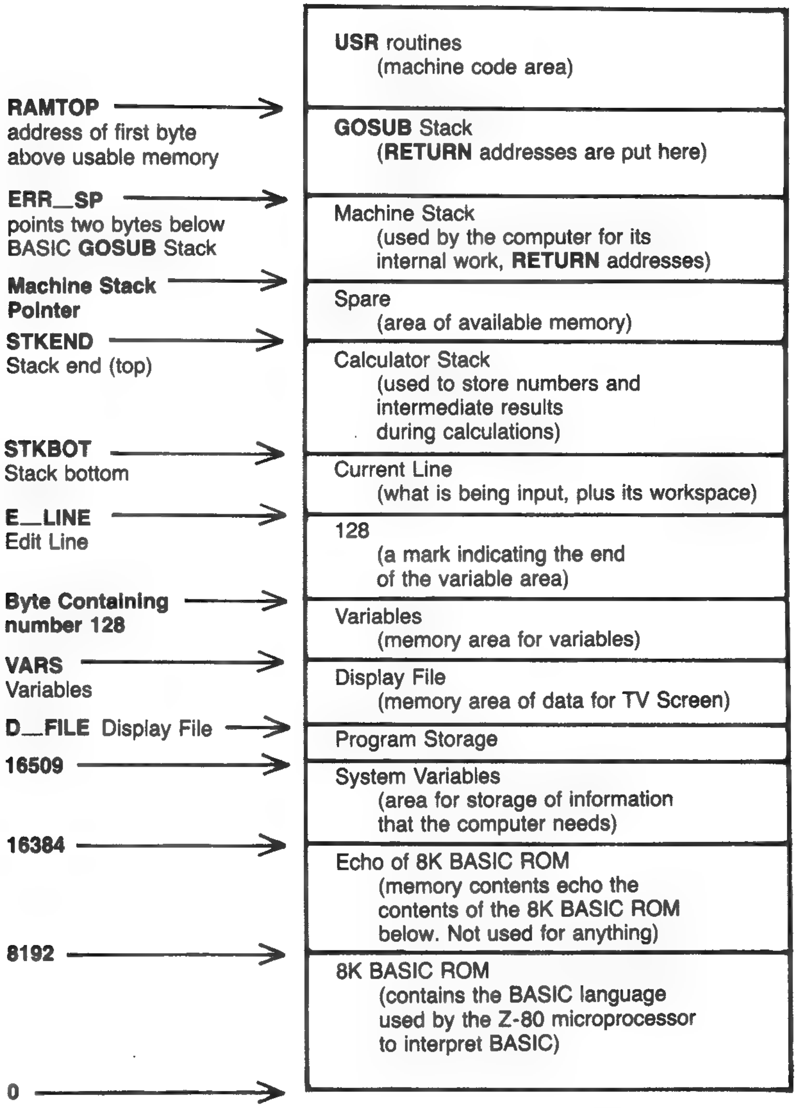

USR routines
(machine code area)
RAMTOP
address of first byte above usable memory
GOSUB Stack
(RETURN addresses are put here)
ERR_SP
points two bytes below BASIC GOSUB Stack
Machine Stack
(used by the computer for its internal work, RETURN addresses)
Machine Stack Pointer
Spare
(area of available memory)
STKEND
Stack end (top)
Calculator Stack
(used to store numbers and intermediate results during calculations)
STKBOT
Stack bottom
Current Line
(what is being input, plus its workspace)
E_LINE
Edit Line
128
(a mark indicating the end of the variable area)
Byte Containing number 128
Variables
(memory area for variables)
VARS
Variables
Display File
(memory area of data for TV Screen)
D_FILE Display File
Program Storage
16509
System Variables
(area for storage of information that the computer needs)
16384
Echo of 8K BASIC ROM
(memory contents echo the contents of the 8K BASIC ROM below. Not used for anything)
8192
8K BASIC ROM
(contains the BASIC language used by the Z-80 microprocessor to interpret BASIC)
0

Figure 6-2
Memory map of the T/S 1000

standard 2K T/S 1000. You will get other numbers for different amounts of RAM.

How Much You've Got

You can also use **PEEK** to see how much memory remains in the different memory areas. Figure 6-2 shows the memory map of the T/S 1000. BASIC language resides in ROM memory locations Ø through 8191. There is no physical memory from 8192 to 165Ø9, and peeking in this area just produces an "echo" of the contents of Ø through 8191. Various system information is kept in the system variables area of 16384 to 165Ø8. The next area is program storage. The variables area floats above the display file, and the display file floats above the program area. If you have the 16K RAM Module, then the display file is normally maintained at its maximum size. In this case, the variables area is 25 bytes (for the **ENTER**'s) and 24 lines x 32 columns above the beginning of the display file.

In Figure 6-2, the term *stack* means an area of memory that is accessed sequentially. Items are stored and retrieved from the top of the stack at STKEND, addresses 16412 and 16413. As you saw in the chapter on subroutines, stacks are a common way of storing and retrieving information on a last-in, first-out basis.

Peeking the variables area is tricky if you only have the standard 2K T/S 1000. The address of the beginning of the variables area is contained in addresses 164ØØ and 164Ø1. The problem in viewing the variables area is that its beginning is after the end of the display file. Each time you print on the screen, the display file increases. This increase causes the start of the variables area to move because it floats on top of the display file.

If you have the 16K RAM Module, the display file generally stays at its maximum size, making it easier to find.

To see the variables area, you must peek at locations 164ØØ and 164Ø1 just before printing. The program below shows how this is done. Again, be sure to enter all program lines exactly as shown.

```
 5 REM PEEK
1Ø PRINT "LOWER=?";
2Ø INPUT L
3Ø PRINT L
4Ø PRINT "UPPER=?";
5Ø INPUT U
6Ø PRINT U
7Ø FOR I=L TO U
8Ø PRINT I+PEEK 164ØØ+256*PEEK 164Ø1;TAB
8;PEEK (I+PEEK 164ØØ+256*PEEK 164Ø1)
9Ø NEXT I
```

For this program, let the lower L be Ø, and the U be the number of memory locations -1 that you want to see. Before you run the program, insert the test lines

```
1 LET A=1Ø
2 LET A1=1Ø
```

and then run your program for L=Ø, U=1ØØ. For the 2K T/S 1000, you will see the numbers shown below. If you have more than 2K RAM, your addresses will be different but the contents will be the same.

```
16766     1Ø2
16778     132
1679Ø     32
168Ø1     Ø
16811     Ø
16821     Ø
16831     166
16843     157
16855     132
16867     32
```

```
16878     Ø
16888     Ø
16898     Ø
169Ø8     113
1692Ø     Ø
1693Ø     Ø
1694Ø     Ø
1695Ø     Ø
1696Ø     Ø
1697Ø     122
```

The addresses increase in relation to the number of characters displayed on each line (plus an **ENTER**) because the variables storage area floats above the display file. As more data is displayed, the beginning of the variables storage area increases by the number of characters added.

Notice that there's a 132,32,Ø,Ø,Ø at addresses 16778 through 16821 and 16855 through 16898. As you saw earlier, these are the computer's normalized binary representation of 1Ø.

The 1Ø2 at address 16766 corresponds to the variable A in line 1. The computer adds 64 to the code for A,38, to give 1Ø2. This addition of 64 means that A is a single variable name.

For more than a single letter name, the computer adds 128 to each character. Thus, for A1 you have

A: 38+128=166

1: 29+128=157

Now try your program with

```
1 LET A11=1Ø
```

and you will see

```
16768     166
1678Ø     29
```

16791	157
168Ø3	132
16815	32
16826	Ø
16836	Ø
16846	Ø

The general method for storing variable names for more than 1 character is to

(1) add 128 to the codes of the first and last characters of the name

(2) just use the standard code of characters between the first and last characters. For example, a variable called AABA is stored as

A	166
A	38
B	39
A	166

Now you can appreciate the amount of memory it takes to store even simple variables or numbers. You can cut down on the amount of memory in programs by using the predefined constants provided by the computer in place of numbers. For example, for maximum economy, use

Instead of	Use
LET A=1	**LET** A=**PI/PI**
LET A=Ø	**LET** A=**NOT PI**
LET B=Ø	**LET** B=A
LET C=1	**LET** C=A

where it is assumed that B and C are defined after A, so that they may be defined in terms of A.

The only disadvantage of using predefined constants is that the programs are harder to understand.

With **PEEK**, you can see how much memory different parts of the computer use. For example, to determine

(1) bytes used in program storage:

Beginning of Display File - Start of Program Area
= **PRINT PEEK** 16396+256***PEEK** 16397-165Ø9

(2) bytes used in storage, display, and running the program:

Program + Display File + Variables + Mark byte
= Beginning of E_line - Start of Program Area
= **PRINT PEEK** 164Ø4+256***PEEK** 164Ø5-165Ø9

(3) approximate number of bytes remaining:

Machine pointer stack - stack end

Since there is no system variable that accesses the machine pointer stack, let's use the top of the **GOSUB** stack (ERR_SP) as an approximation. If you aren't using many **GOSUB**'s and nested **FOR-NEXT** loops, then ERR_SP should be within a few bytes of the machine stack pointer,

= ERR_SP - STKEND

= **PRINT PEEK** 16386+256***PEEK** 16387-**PEEK** 16412-256***PEEK** 16413

For more information, see the Appendix on System Variables.

Saving Pictures

You can save any computer pictures displayed on the screen by using **PEEK**. However, if the whole screen is to be saved, you will need 7Ø4 bytes to store the picture in a string variable. If you only have the standard 2K T/S 1000, storage can be a problem. When the program that drew the picture is over 1K bytes, such as the AutoDraw program, the program, plus display file, plus the string exceeds over 3K bytes—which is beyond the capacity of the computer.

Let's see how you can save a picture by using a small program to make the drawing. If you have a 16K RAM Module, you will have enough memory to use the same technique to save pictures for Autodraw and the Plot program.

The following program generates a maze on the screen. Enter and run

```
 5 REM MAZE
1Ø RAND
2Ø FOR I=Ø TO 15
3Ø FOR J=Ø TO 15
4Ø PRINT CHR$ (3+(2 AND RND <.5));
5Ø NEXT J
6Ø PRINT
7Ø NEXT I
```

The **RAND** function in line 1Ø always generates a different starting point, or *seed,* for the random number function, **RND**. The control variable I determines the rows, and J controls the number of columns printed. If you have more than 2K of memory, you can make a bigger maze. Line 4Ø determines whether a horizontal bar character of code 3 (see the Appendix on Character Codes) or a vertical bar character of code 5 is printed.

For example, if **RND**<.5 is true, then

RND <.5=1

thus

2 **AND** 1=2

and

3+2=5

Therefore, line 4Ø does a **PRINT CHR$** (5). Now, if **RND** is greater than or equal to .5, then

RND <.5=Ø

thus

```
2 AND Ø=Ø
```

and

```
3+Ø=3
```

Therefore, line 4Ø does a **PRINT CHR$** (3) and prints a horizontal bar.

After you run this program, your screen will contain a maze. You can save this picture in one of three ways:

(1) By inserting the line

```
8Ø SAVE "PICTURE"
```

This will automatically save the program and all the variables, including the display file where the picture is stored. Note that if you manually enter

```
SAVE "PICTURE"
```

and then reload the program, you will just get a blank screen.

(2) By peeking the contents of the display file to a string variable. This string variable would then contain the data needed to reconstruct the picture.

(3) By copying the picture to a string variable as the picture is being made.

To demonstrate peeking, let's examine how a picture is saved using method 2.

In the following utility program, notice that all its lines are numbered at about the highest you can store. If you want to save any programs and pictures by this method, enter these lines first. You should also store this utility program on tape, so that you won't have to type in the program each time you want to include it with another program. First, load the Screen Save and Load program into memory from tape, and then enter the rest of your program.

```
98ØØ REM SCREEN SAVE AND LOAD
```

```
9810 LET L=PEEK 16396+256*PEEK 16397
9820 LET U=PEEK 16398+256*PEEK 16399
9830 LET S$=""
9840 FOR I=U TO L STEP -1
9850 LET S$=S$+CHR$ PEEK I
9860 NEXT I
9870 SAVE "PICTURE"
9880 STOP
9900 FOR I=U-L TO 1 STEP -1
9910 IF CODE S$(I)<>118 THEN GOTO 9940
9920 PRINT
9930 GOTO 9950
9940 PRINT S$(I);
9950 NEXT I
9960 STOP
```

If you run this program in **SLOW** mode, you will see a maze being printed. After it is plotted, the **SAVE** will start in about 30 seconds. Turn on the recorder about 20 seconds after the plot stops and record. The program will **SAVE** and then **STOP** at line 9880 with a **STOP** report of 9. Delete the program and then load the program back in. You will see the same picture as you did when the program stopped before. The report code of 9/9880 will also appear because it represents the condition of the Maze program when it was saved.

The data for the picture is stored in the variable S$. Do a

```
PRINT S$
```

to see the contents of S$.

Notice that the picture doesn't look like the original maze. It is a rectangle with lots of question marks in it. The question marks are the **ENTER**'s at the end of each line in the original maze. You didn't see them in the original picture, because the **ENTER**'s weren't printed. S$ contains the data on the screen, plus the

ENTER at the end of each line. If you want to save a complete screen of 7Ø4 characters, one way is to include the 25 extra characters for the **ENTER**'s, as done in this program. To reconstruct the maze from the data in S$, do a

```
GOTO 99ØØ
```

and you will see the original maze reconstructed.

In this program, line 981Ø peeks at locations 16396 and 16397 to determine the beginning of the display file, and assigns it to a variable, L, for convenience. Line 982Ø looks in locations 16398 and 16399 to find the current **PRINT** position. This position indicates where the next item would be printed after the maze is complete. Line 983Ø initializes S$ to the null string. The loop of lines 984Ø through 986Ø starts peeking from the current print position back to the beginning of the display file. As the loop peeks, the computer stores the character in S$ by concatenating it. In addition, because we're going backwards, S$ is also the reverse of the maze you drew using **PRINT** S$. Although it is possible to use a numeric dimensioned variable, say S(I), such a variable uses up a lot more memory in storage. After the loop, line 987Ø saves the picture.

The loop of 99ØØ through 995Ø reconstructs the picture. If an **ENTER**, code 118, is found, then the program executes the **PRINT** of 992Ø. If it does not find an **ENTER** code, then the program prints the character.

Once the display file is stored as a string variable, you have control over it through standard BASIC commands. For example, change line 994Ø to

```
994Ø PRINT CHR$ 1Ø AND S$(I)=CHR$ 3;
```

and add

```
9945 PRINT CHR$ 133 AND S$(I)=CHR$ 5;
```

Line 994Ø substitutes a **CHR$** 1Ø whenever S$=**CHR$** 3, the horizontal bar.

Line 9945 substitutes a **CHR$** 133 whenever S$=**CHR$** 5, the vertical bar.

Do a **GOTO** 99ØØ, and you will see a very different looking maze printed out.

Peeking Ahead

PEEK is very useful in games because it allows the computer to detect collisions. As an example, enter and run the following game. Your spacecraft has entered a region of black holes. How high a score can you get by avoiding collisions? Use the **8** key to move right and the **5** key to go left. A score of 1ØØ is average.

```
  5 REM BLACK HOLES
 1Ø LET A$=“*******”
 2Ø LET X=11
 3Ø LET Y=15
 4Ø LET SC=Ø
 5Ø PRINT AT X,Y;“ ”;AT 21,24*RND;A$(1 TO
 6*RND+1)
 6Ø SCROLL
 7Ø PRINT TAB 31;AT 21,Ø
 8Ø LET SC=SC+1
 9Ø LET I$=INKEY$
1ØØ LET Y=Y+(I$=“8”)-(I$=“5”)
11Ø PRINT AT X,Y;“V”;AT X+1,Y;
12Ø IF PEEK (PEEK 16398+256*PEEK 16399)<>23
THEN GOTO 5Ø
13Ø PRINT AT X+1,Y-3;“SCORE=”;SC
```

Line 1Ø defines A$ to contain the “black holes.” Lines 2Ø and 3Ø initially set the spacecraft to about the middle of the screen. Line 5Ø blanks out the previous position of the spacecraft, to keep it from scrolling up the screen with the black holes. The rest of line 5Ø plots from 1 to 7 holes at a random point on the last display line. Notice that the **INT** function is not used to round off the argument, 1 **TO** 6***RND**+1, of A$ in line 5Ø. The **TO** argument is automatically rounded off. The 6***RND** will generate Ø to 6 black holes. By adding 1, there are 1 to 7 black holes randomly generated.

Line 6Ø scrolls the screen up, and line 7Ø is added to give the display more stability as it scrolls. Line 7Ø pads out the bottom scrolled lines with all blanks and then resets the print position to the beginning of the line. Try running the program without line 7Ø, and you will see that the display looks rather erratic. Line 8Ø increments the score, SC, after a new **SCROLL** cycle. Lines 9Ø and 1ØØ move the Y position of the spaceship as the **8** and **5** keys are pressed. A longer way of writing line 1ØØ is

```
1ØØ IF I$="5" THEN Y=Y-1
1Ø5 IF I$="8" THEN Y=Y+1
```

With line 1ØØ, if I$="8," then the logical relation I$="8" is true, and so Y=Y+1. If I$="5," then the logical relation I$="5" is true, and so Y=Y-1.

Line 11Ø prints the spaceship at the new Y position and then sets the current print position directly under it. Line 12Ø peeks at this current position by looking at addresses 16398 and 16399. These addresses contain the information of the current printing position. The first **PEEK** of line 12Ø finds the value stored in the address. If the value is 23, then an * is stored there, and the computer goes to line 13Ø and prints the score. If it is not an *, the computer goes back to line 5Ø.

Notice that there is no error checking in this program. If you go too far left with the **5** key, the spaceship is reflected back to the right. If you go too far right, the program crashes. Adding error checking would slow down the game. A screen boundary, however, could be drawn initially to mark the limits of the playing area. Enhancements can be added to the Black Hole program, such as another variable to keep track of the highest score registered. In addition to keeping track of collisions, you can give the spaceship missiles to use in blasting its way through the black holes. You can also peek ahead of the missile to see if it hits a black hole.

Poking Around

Just as you can peek into memory locations, you can also *poke* into them if they are RAM. Generally, the term RAM indicates the following:

(1) You can access any memory byte in about the same length of time. This is why this kind of memory is called *random access memory,* as opposed to the *sequential access memory* of magnetic tape. Suppose you want a program at the end of a tape, and the tape is completely rewound. You must move the tape forward until it is at the program of interest. With RAM, it doesn't matter where the physical location of the data is. You don't have to read through all the data in front to get to the program you want.

(2) You can both read data from RAM and write data into RAM. If you have *read only memory (ROM),* you can only read data from it, as the name implies. The BASIC in your T/S 1000 is contained in ROM. You can't write into it. However, ROM has one advantage—that its contents aren't lost if power is removed. In fact, that's another reason why BASIC is put in ROM. Generally, most types of RAM made from semiconductor material today lose their contents when power is removed.

The **POKE** command lets you put data into any RAM address. **POKE** is a keyword located on the **O** key. If **POKE** is not used correctly, it can cause your program to crash. The **POKE** command is extremely powerful, because with it you can alter memory locations that the other BASIC commands keep you from harming. To utilize **POKE** fully, you should know how the T/S 1000 BASIC operates, and also know machine-language programming for the Z80 microprocessor. A good reference for finding out what's in BASIC is *Understanding Your ZX-81 ROM,* by Dr. Ian Logan, published by the The Essential Software Company, 47 Brunswick Centre, London, W1 CN 1AF.

As a demonstration of **POKE**, let's modify the Screen Save and

Load program so that we can tell when each character is being peeked. Add lines 9852, 9854, and 9856 to the Maze program with Screen Save.

```
9852 PRINT "3";
9854 POKE 16398,I-256*INT (I/256)
9856 POKE 16399,INT (I/256)
```

You should also change line 9940 to

```
9940 PRINT S$(I)
```

and delete line 9950 to get the ordinary printing.

Now do a **RUN** to start the program over again from the beginning. You will see the maze being constructed, and then being eaten up by the 3's. Notice that the first 3 that appears is on the line after the maze. This is the current printing position. Then the 3's start eating away at the display file as the display file gets peeked into S$. After the 3 gets to the beginning of the file, the program goes into its **SAVE** mode. When the **SAVE** is done, do a **GOTO** 9900, and you will see the original maze.

Lines 9854 and 9856 keep decrementing the addresses 16398 and 16399 of the current print position. The 3 keeps getting printed towards the left and up, as the current printing position goes toward the beginning of the display file.

POKE can be used to bypass the normal processing of the computer. For example, enter and run this program:

```
10 PRINT 1
20 GOTO 10
```

You will see a column of 1's going down the screen until it is full.

Now let's poke the second byte of the program area by using **POKE** 16510,0. This byte contains the low byte of the line number data. As we saw before, the line number of the first statement of a program is

```
256*PEEK 16509+PEEK 16510
```

List this program, and you will see

```
 Ø PRINT 1
2Ø GOTO 1Ø
```

By poking a Ø into the low byte of this line number, you bypassed the normal input restrictions and entered a line Ø. If you change line 2Ø to

```
2Ø GOTO Ø
```

you can run the program just as before.

Now change line 1Ø to

```
1Ø PRINT 1Ø
```

and list the program. You should see

```
 Ø PRINT 1
1Ø PRINT 1Ø
2Ø GOTO Ø
```

Do a **POKE** 1651Ø,1Ø and list. Now you will see

```
1Ø PRINT 1
1Ø PRINT 1Ø
2Ø GOTO Ø
```

By poking a 1Ø back into the line number, you've managed to bypass another of the computer's restrictions. Normally, you can't have two lines with the same number. However, **POKE** bypasses the normal input processing. You can run this program and get a normal output of 1,1Ø,1,1Ø,1,1Ø... until the screen fills.

Now do a **POKE** 165Ø9,2Ø and you will see

```
513Ø PRINT 1
  1Ø PRINT 1Ø
  2Ø GOTO Ø
```

The first line 1Ø was changed to 513Ø because 256*2Ø+1Ø=

513Ø. Notice that line 513Ø does not follow line 2Ø because it was not input normally. You can run this program and still get 1,1Ø,1,1Ø...

Now do a **POKE** 165Ø9,4Ø. This should generate a line number greater than the 9999 allowed by BASIC. The line number should appear as

256*4Ø+1Ø=1Ø25Ø

But what you see listed is

```
A25Ø PRINT 1
  1Ø PRINT 1Ø
  2Ø GOTO Ø
```

Actually, the line number A25Ø is the same as 1Ø25Ø. However, it is expressed in a mixture of letters and numbers. The letters and their decimal equivalents are

A	1Ø
B	11
C	12
D	13
E	14
F	15
G	16

You can see this mixture by poking in different values at address 165Ø9.

If you substitute the decimal 1Ø for A in line A25Ø, you will get 1Ø25Ø. Normally, line numbers above 9999 are not input; therefore, BASIC does not have to handle them. As before, you can still run this program. Now do a

POKE 165Ø9,63

POKE 1651Ø,255

and list. You will see the first line number as G383. The decimal number is

```
63*256+255=16383
```

If you try increasing this number by one more,

```
POKE 16509,64
POKE 16510,0
```

you won't be able to list the program. The part of BASIC that reads and prints line numbers can't handle anything this big. As a result, your program becomes invisible. You can't run or list it, but it is still there. Do a **POKE** 16509,0 and a **POKE** 16510,10 to get back to the original version. Now you can list the program.

```
10 PRINT 1
10 PRINT 10
20 GOTO 0
```

To make your program invisible, but still runable, enter the following test program:

```
 1 REM
10 PRINT "HELLO"
```

Now do a **POKE** 16513,118 to replace the **REM** token with an **ENTER**. If you do a list, you will see only a "1." The rest of the program has become invisible because the **ENTER** is illegal. Normally, you cannot enter a line with just an **ENTER**. You can still run the program, however, even though it is invisible. To see the program again, poke back the original REM with a **POKE** 16513.

Renumbering

As a practical example of **PEEK** and **POKE**, let's look at a program that renumbers the lines of another program. This program is useful when you need to insert a line and can't squeeze it in. It also helps your programs look neat by uniformly spacing the lines in increments of 10. With renumbering, you can use part or all of an existing program in a new one. You can renumber the lines of the old program or subroutine you want to use so that they can be incorporated where you want them in the new program. Once you have inserted the lines, you can start entering the rest of the new program lines.

The renumbering program below does simple renumbering of program lines. It only renumbers the line numbers at the beginning of lines. A more sophisticated version would also change line numbers in **GOTO**'s and **GOSUB**'s. The input data is

OLD START - the first line number you want to renumber

OLD FINAL - the last line number you want to renumber

NEW START - the new starting line number

INCREMENT - how much should be added to each line

For example,

```
Old Program            New Program
5 LET A=1              1Ø LET A=1
7 LET B$="DICK"        2Ø LET B$="DICK"
2Ø PRINT A,B$          3Ø PRINT A,B$
```

Old Start=5, Old Final=2Ø, New Start=1Ø, and Increment=1Ø will give you a New Program using the Old Program. Enter the Old Program, shown above, and the Renumber program below.

```
977Ø REM RENUMBER
978Ø PRINT "OLD START,OLD FINAL,NEW START,
INCREMENT="
979Ø INPUT OS
98ØØ INPUT OF
981Ø INPUT NS
982Ø INPUT IN
983Ø LET P=165Ø9
984Ø LET LINE=256*PEEK P+PEEK (P+1)
```

```
985Ø LET LENGTH=PEEK (P+2)+256*PEEK
(P+3)+4
986Ø IF OS=LINE THEN GOTO 989Ø
987Ø LET P=P+LENGTH
988Ø GOTO 984Ø
989Ø POKE P,INT (NS/256)
99ØØ POKE P+1,NS-256*INT (NS/256)
991Ø LET P=P+LENGTH
992Ø LET LINE=256*PEEK P+PEEK (P+1)
993Ø LET LENGTH=PEEK (P+2)+256*PEEK
(P+3)+4
994Ø LET NS=NS+IN
995Ø IF OF=LINE THEN GOTO 997Ø
996Ø GOTO 989Ø
997Ø POKE P,INT (NS/256)
998Ø POKE P+1,NS-INT (NS/256)
999Ø PRINT "DONE"
```

Then run the program using a **RUN** 978Ø command for the Old Program data shown before. When the Renumber program stops, it will print "DONE." Do a **LIST** and you will see the New Program.

Lines 978Ø through 982Ø input the data. To conserve memory, the input data is not printed back on the screen. Line 983Ø sets the start of the program's search to the beginning address of the program area. Lines 984Ø through 988Ø search for the Old Start line. Line 984Ø peeks at the line number of every line and stores it in LINE. The length of each line, plus 4, is stored in LENGTH. The 4 is added to get the total number of characters in each line. The length stored in the line does not include the 2 bytes for line number and the 2 bytes for the length itself. The variable P always points to the beginning of the line—that is, P points at the first byte of the 2-byte line number. In order to point at the first byte of the next line, you have to add the length of the line to P, as in line 987Ø. Line 986Ø checks whether the Old Start line equals the line number currently being peeked at. If so, the loop of lines 984Ø through 9880 is executed to line 9890. Lines 9890 and 9900 poke in the New Start line number.

The length of the line is added to the pointer P, and the line number and length of the next line are found. The new line number we want to poke is calculated in line 9940 by adding the increment IN to the previous line number NS. Line 9950 checks whether the Old Final equals the latest line peeked at. If so, the loop of lines 9890 through 9960 is exited, and the final line number poked. If it is not the final line number, then line 9960 keeps going back to line 9890 for more renumbering. Notice that there is no error checking done by this program. If you give a line number that is not in your old program, the renumber program will get caught in a loop searching for it. You may want to add enhancements like error checking and changing line numbers of **GOTO**'s and **GOSUB**'s.

Machine Code Programming

Although BASIC is convenient for many applications, there are cases in which programming at the machine level is better. For example, machine code executes much faster than BASIC because the computer does not have to interpret each line. You can call the machine code to execute by using the BASIC command **USR**, which is a shifted function located under the **L** key. The machine code can then be executed for those portions of the program that require its speed. The disadvantage of machine code is that it takes longer to program an application and debug it, compared to the time it takes with a *high-level language* such as BASIC. High-level languages are closer to the way people think, thus making them easier to use.

It is not possible to teach you how to program in machine language in this one seciton, but we can show you *how* to use machine code programs. Magazines such as *Sync* and *Syntax* regularly feature machine code programs. You will find these programs very interesting and useful.

Machine code programs should be stored where they won't be affected by the BASIC program; otherwise, they may be overwritten or moved. If either happens, the computer will find it impossible to execute correctly a machine code routine by the **USR** function. The **USR** function must call the machine code routine at a fixed address. If the machine code moves, the **USR** can't find it. One way of storing machine code programs is to move RAMTOP to a point lower than the actual end of memory. Then the machine code program can be stored at that point because even the **NEW** command never clears memory above RAMTOP.

Suppose you have a machine code program that is 14 bytes long and you want to store it above RAMTOP.

First, let's see where RAMTOP points. Try

```
PRINT PEEK 16388+256*PEEK 16389
```

On the standard 2K T/S 1000, you will get 18432. This is actually 1 byte past the end of physical memory. To reserve 14 bytes, we want RAMTOP to be

18432−14=18418

Therefore,

```
POKE 16388, 18418−256*INT (18418/256)
POKE 16389, INT (18418/256)
```

will reserve 14 bytes at the top of memory. You can put anything into the upper 14 bytes, and it will not be affected by the BASIC **NEW** command. However, it may be affected by the machine stack. Therefore, you should reserve more than the 14 bytes of memory, say 60 bytes, to be on the safe side.

If you check RAMTOP, you will see that it is indeed 18418. Poke the original RAMTOP back as

```
POKE 16388, Ø
POKE 16389, 72
```

for the standard 2K computer, and enter the following program for machine code loading and execution:

```
 5 REM MACHINE LOADER
1Ø LET R=PEEK 16388+256*PEEK 16389
2Ø PRINT "BYTES TO LOAD=?";
3Ø INPUT B
4Ø PRINT B
5Ø LET R=R-6Ø
6Ø POKE 16388, R-256*INT (R/256)
7Ø POKE 16389, INT (R/256)
8Ø FOR I=Ø TO B-1
9Ø PRINT I+1; TAB 5; "CODE=?";
1ØØ INPUT C
11Ø PRINT C
12Ø POKE R+I,C
13Ø NEXT I
14Ø PRINT "DONE LOADING"
15Ø CLS
16Ø RAND USR R
```

Line 10 determines the end of physical memory +1, RAMTOP. If you have the 16K or 2K, or any other RAM, this line will automatically determine what RAMTOP is. Line 5Ø subtracts 6Ø bytes from RAMTOP. This number is poked as the new RAMTOP in lines 6Ø and 7Ø. Lines 8Ø through 13Ø poke the machine language into the space reserved above RAMTOP. Line 15Ø executes the machine language by using the **USR** command (pronounced "user"). The **USR** must always be followed by the starting address of the machine code to be executed.

The **RAND** in front of **USR** in line 16Ø is actually a dummy command. The machine language could also be

```
16Ø LET A=USR R
```

where A is any variable name.

The advantage of **RAND** over **LET** A is that **RAND** uses several bytes less than **LET** A.

Now run this program for the 2K T/S 1000 only, using the following data. Enter 14 for the number of bytes, and then the following as the machine code: 62, 22, 14, 32, 215, 13, 194, 2ØØ, 71, 61, 194, 198, 71, and 2Ø1. You should see the screen fill quickly with different characters. The top line should be all "–" signs, the next line "+" signs, the third line "=" signs, and so on down to horizontal bars on the bottom. If you look in the Appendix of Character Codes, you will see that this program displays only the characters for codes from 22 to 1.

Notice how fast the display was printed, even in **SLOW** mode. It would be even faster in **FAST** mode. Now poke the original RAMTOP back, as **POKE** 16388,Ø and **POKE** 16389,72, and try the following variation on this program. Enter these numbers for 16 bytes: 62, 31, 6, 22, 14, 32, 215, 13, 194, 2Ø2, 71, 5, 194, 2ØØ, 71, and 2Ø1. Now you will see a pattern of only one type. The character that's printed is determined by the second byte entered. In this case, the 31 determined that the character of code 31 would be printed.

You can facilitate character selection by adding the lines

```
132 PRINT "CHARACTER=?";
134 INPUT C$
136 POKE R+1,CODE C$
```

to your program, and then **GOTO** 132 to input it, since all the data is still stored in memory. Each time the program is run, you should poke back your original RAMTOP. If you don't, your RAMTOP will keep going down.

In another technique, the machine code is put into a **REM** statement at the beginning of the program. Enough space is reserved after the **REM** to hold the machine code. Then a **USR** is made to the memory after the **REM**. The advantage of adding this **REM** is that you can save the machine language code as part of your program.

Chapter 7
Expanding Your Horizons

Up to this point, we have discussed the fundamentals of programming in BASIC. By studying the material in the two volumes of this text and trying the examples, you should have developed by now a good grasp of the BASIC programming language. You should also be able to understand magazine articles and other current literature on programming. Computer magazines, such as *Sync*, *Syntax*, *Compute!*, and *Byte*, are a good source of new and useful information. In addition, you may want to take computer courses at a local college or university. Most schools offer both credit and noncredit courses that can help you learn more about computers.

Another way you can learn about computers is by expanding your computer. There are many software and hardware enhancements you can add to the T/S 1000 to increase greatly its power and capabilities.

For more information about computers, see *Foundations of Computer Technology*, and *Modern Computer Concepts*. These books provide a thorough introduction to computer history and technology. The other books in that series—*BASIC: Fundamental Con-*

cepts and *BASIC: Advanced Concepts*—cover programming in two other popular dialects of BASIC.

One dialect of BASIC, produced by Microsoft, Inc., is used on many microcomputers. Another version of BASIC, designed by Digital Equipment Corporation (DEC), is available on its popular computers. Many similarities exist between the version of BASIC that you have learned and Microsoft and DEC BASICs. By learning the similarities and the differences among dialects of BASIC, you can adapt programs written for other types of computers to run on your T/S 1000.

Hard Copy

If you have ever tried to debug a program that was longer than one screen, you have probably wished you could see the entire program rather than just 22 lines. No doubt, there have also been times when you have wished for a printed listing or output of your program. For these and many other reasons, it's nice to have a device that can produce hard copy from the computer. *Hard copy* refers to a permanent, printed record of your program or its output. A device like your TV screen is a *soft-copy* device because the image on it disappears when power is removed. In other words, the image is not permanent.

A variety of hard-copy devices is available. One of the most useful is a *printer*, which is a device for producing hard copy from a computer using one of a number of printing techniques.

Thermal printers use heat to produce hard copy. The printer's printhead moves across a special type of paper that is sensitive to heat. A group of needles on the printhead is heated by an electric current. The computer sends a code for the character to be printed to the printer. The printer translates this code into the appropriate electronic signals to heat selected needles on the printhead. These needles form the character when pressed against the special thermal-sensitive paper.

This general technique of activating needles to form characters is called *dot-matrix printing*. A wide range of characters can be printed with the dot-matrix technique, which is much faster than ordinary typing. For example, some dot-matrix printers can print hundreds of characters per second (cps).

In contrast, another type of dot-matrix printing, called *impact dot-matrix printing*, presses selected needles against an inked ribbon to form characters. This type of printer is noisier than the thermal, but has the advantage of being faster. In addition, the impact printer uses ordinary paper rather than the special paper required for the thermal printer. In fact, the term "dot matrix" usually refers to the impact dot-matrix printer.

Impact dot-matrix printers are also capable of printing multipart forms consisting of alternating layers of paper and carbon paper. The impact of the printing needles allows multiple copies to be made. Depending on the printer, up to six copies may be produced at one time.

Some impact printers have multicolored ribbons for color printing. Others have a single ribbon in a cassette designed for easy removal and insertion. Color printing can be done with this type of printer if you manually remove and insert the appropriate color ribbons, then have the computer output again. Of course, you must have the appropriate software to control the printing process after each ribbon change.

If you look at the print quality of a dot-matrix printer, you will see that it is not as good as a typewriter because each character is composed of dots. However, some dot-matrix printers are designed to achieve better print quality by overlapping the dots to produce a more solid character. In fact, some printers can be switched to provide either fast, standard dot-matrix characters, or slower, *correspondence-quality* characters. The term "correspondence quality" refers to characters that have good enough resolution for use in business correspondence.

For true correspondence quality, a printer with *solid fonts* is the standard. The term "solid font" means that each printing element, or font, is made in a single piece. One example of a solid-font printing device is a typewriter. Each character has a separate, solid font that makes a printed character when the font impacts the ribbon against the paper. In contrast, dot-matrix printers synthesize each character by impacting or heating needles in the printhead.

The *daisywheel* is a common type of solid-font printer whose name is derived from the fact that its printing elements are arranged like the petals of a daisy around a central hub. In operation, the hub spins rapidly as it moves across the paper. The appropriate printing element is pressed against the inked ribbon to produce a character on the paper. Daisywheel printers are faster than ordinary typewriter-style printers. Typically, daisywheel printers can do 45 or more cps, compared to the 15 or 30 of a typewriter type of printer.

A *plotter* is another kind of hard-copy device that uses a ball-point pen, or some other type of pen, to draw lines. The resolution of lines is much finer than dot-matrix characters. Some plotters use different colored pens to draw multicolor plots. Although pen plotters do not draw as fast as dot-matrix printers, the plotters do give the best resolution for graphics.

Many other hard-copy devices are available, such as electrostatic printers and laser printers. These devices are capable of high speed and high resolution. However, they cost considerably more than dot-matrix printers and, therefore, are not commonly found on low-cost computer systems.

Interfaces

Some printers and other devices can be plugged into the back of your computer and they will work immediately. The printer made by Timex is one example. You control the Timex printer with commands such as **LPRINT**, **LLIST**, and **COPY**, which are on the T/S 1000 keyboard.

Unless the printer or other device was designed specifically for the T/S 1000, however, you will need an *interface* to make the device work properly with your computer. The term "interface" refers to an electronic circuit or device that provides the necessary

1. electrical signals
2. physical connectors
3. software

to make the printer or other device work correctly with your computer. You can't just buy any piece of equipment and expect it to work with your computer. All three conditions listed above must be met for your printer or other device to work.

Printers are usually manufactured with one or two *industry-standard interfaces*. The term "industry-standard interface" refers to the set of electrical and physical standards that are accepted by the electronics and computer industry. Your computer must supply the appropriate signals to make the printer work properly.

One industry standard interface is the RS-232-C serial interface. A second type is the Centronics® parallel interface, named after the company that first popularized it. The printer manufacturer may include one or both types in the printer design. Generally, the Centronics interface is preferred for its higher speed. However, you cannot just connect printers with a RS-232-C or Centronics interface to your T/S 1000 because it does not have either type of interface.

To connect a printer with a RS-232-C or Centronics parallel interface, you need to add a standard interface to your computer to provide electrical, physical, and software compatibility. For example, the Centronics parallel interface manufactured by the Memotech Corporation plugs into the back of your T/S 1000 and provides these compatibilities. A cable is also needed to connect the printer to the Memopak printer interface. Using this interface, you can also access directly the **LPRINT**, **LLIST**, and **COPY** commands from the keyboard.

Modems

The term *modem* is a contraction of *mo*dulator/*dem*odulator. A modem allows your computer to communicate with other computers or terminals over a telephone system.

A *computer terminal* is a device that allows a *person* to communicate with a computer. One common terminal uses a cathode ray tube for display and is often called a CRT. A CRT is basically a TV tube and a keyboard. You enter commands and requests through the keyboard to the computer. Output from the computer appears on the CRT screen. A single large computer may support hundreds of CRT terminals. One example of such a system is an airline reservation system where all of the terminals need access to the same information. A bank computer system is another example. In the T/S 1000, the terminal is incorporated into the computer.

A modem extends the range of your computer beyond the horizon. Many companies offer information services that are accessible by computer. For example, Dow Jones provides information about stock prices. By subscribing to their service, you can have your computer show on your TV display the latest price quotations as well as other information. Another application is transferring programs you've written or other data between your computer and a friend's.

Unless a modem is specifically manufactured for the T/S 1000, you will need a RS-232-C serial interface to connect the modem to your computer. In addition, you will need the appropriate software. The type of software you need will depend on its application. For example, if you want to obtain information from another computer or information service, your T/S 1000 must simulate a computer terminal.

Disks

A *disk* is a round, flat platter that uses magnetism to store programs and data. Areas of the disk can be magnetized and demagnetized by the disk head as the disk rotates at several hundred or more revolutions per minute. There are two main types of disks: hard disks and floppy disks.

Hard disks use a rigid platter and store *megabytes* of information. The term "mega" means million. *Floppy disks* or diskettes have a magnetic coating over a thin plastic underlayer, or *substrate*. The floppy disk is so thin and flexible that it literally can flop if not contained in a protective jacket. Floppy disks cannot store as much information as hard disks, but the equipment needed to use floppy disks is simpler and cheaper.

Floppy disks are found on many brands of microcomputers. Depending on its design, a floppy disk can store from 88 Kilobytes (1 Kilo = 1024 bytes) to 1 Megabyte. The standard floppy disk is 8″ in diameter and stores about 256 Kilobytes. Minifloppies are 5 1/4″ in diameter, and microfloppies are about 3″.

The floppy disk is the medium on which information is stored. The disk rotates inside a floppy-disk drive that also contains the electronics to control the *reading and writing of information to the disk*. "Reading" refers to decoding the signals from the disk head as it passes over the disk surface. "Writing" means to transform signals from the computer into the magnetized regions on the disk.

As usual, unless the disk has been specifically made for the T/S 1000, you will need an interface and the appropriate software.

One advantage of disks is that large amounts of information can be transferred quickly and reliably between your computer and the disk. Disks are also preferable to cassette tapes for mass storage of programs and data.

Because of their large capacity and speed, disks can store other computer languages and utility programs. Rather than being restricted to BASIC, your computer can run any computer language if the appropriate hardware and software are available.

Keyboards

Some companies, such as Gladstone Electronics, provide standard typewriter-style keyboards for your T/S 1000. In some designs, the T/S 1000 fits completely inside the new keyboard. In others, the T/S 1000 is connected to the keyboard by a cable. These keyboards are convenient if you prefer the standard typewriter keyboard. A number of enhancements are also provided in some designs, including audible clicks when a key is depressed and auto repeat. "Auto repeat" means that multiple characters are produced when the key is held down.

RAM Expansion

Additional RAM is another useful hardware enhancement. Companies such as Gladstone Electronics and Memotech Corporation offer large-capacity RAM packs that plug into the back of your T/S 1000. These packs are available in 16K, 32K, and 64K sizes, which allow larger programs and data to be stored in memory.

Assemblers

If you plan to do much machine-code programming or just want to learn it, then an assembler is an indispensable software tool. An *assembler* is software that you load into your T/S 1000 to facilitate writing machine code. You write your program in mnemonics, which are abbreviations of the English equivalents for the machine-language instructions. The assembler then translates the mnemonics into the actual machine-language code. It is easier for a programmer to use mnemonics of English words than to use numbers. Another advantage of an assembler is that you can easily assign the final code to the specific region of memory that you want.

The Arctic assembler from Gladstone Electronics is one example of an assembler. It allows you to inspect memory locations for their contents and has other enhancements. Another assembler is available from Bob Berch Software.

Compilers

A *compiler* is a program that translates a high-level language into a lower one. For example, a compiler can translate a BASIC program into the machine code to run on your T/S 1000. The original program to be compiled is called the *source program*, and the output of the compiler is called the *object program*, or *object code*. Some compilers are designed to translate source code into assembly language, and others translate into machine code that can be executed directly by your computer.

The advantage of a compiler over the BASIC interpreter in your T/S 1000 is that a compiled program runs much faster. Depending on the compiler and your BASIC program, the compiled program may execute from 5 to 100 times faster than your interpreted BASIC program. The interpreter in the T/S 1000 translates each statement into machine code, executes it, then goes on to interpret the next line. In contrast, a compiler translates the entire BASIC program into machine code first, which saves time because each line does not have to be interpreted.

Some compilers restrict the type of statements they can accept. For example, the Integer Compiler available from Bob Berch Software and the M Coder from Personal Software Services compile programs with only integer arithmetic. That is, these compilers will not handle floating-point numbers.

Although an integer compiler is a restriction in programs that use floating-point arithmetic, it does speed up program execution. A compiled program also takes up less space in memory because the variables and numbers in the program store only integers. Floating-point compilers generate larger object code because the numeric variables and numbers must also allow space for the decimal representation of numbers.

A compiler offers one of the advantages of assembly language—speed—without your having to learn assembly language. Generally, much less time is needed for someone to program an application in a high-level language rather than assembly language. In fact, that is one reason why compilers were developed. A compiler would be a great help in speeding up the execution of games. The main advantage of an assembler is that you have total control over every instruction. If you are limited by memory size, then an assembler allows you to pack your code more tightly into the available memory.

Games and Utilities

Although the T/S 1000 has only been on the U.S. market a short time, much software is available, with the promise of more to come. The following is a sampling of some of the programs currently available from Softsync, Incorporated. The amount of memory required is also indicated.

Programmers' Toolkit —16K

This cassette tape contains a number of *utility programs.* A utility program is one that helps you program more efficiently. The programs on this tape include:

1. RENUMBER—renumbers lines in a BASIC program and automatically renumbers GOTO and GOSUB lines.
2. SEARCH AND REPLACE—replaces all occurrences of a character you indicate in your program with any other character you specify. As an example, you might want to change the variable name X1 to X2 throughout a program.
3. SEARCH AND LIST—allows you to search your program for a specific character. The line numbers that the character appears on will then be printed.
4. HYPER-GRAPHICS MODE—changes the starting address of the character table.
5. FILL—enables you to fill a specified number of lines with any character you specify.
6. REVERSE—lets you change the characters in any specified lines to inverse video.
7. FREE—indicates how many bytes of memory are still free (available for use).

Graphics Kit —16K

The Graphics Kit is similar to the Programmers' Toolkit in that it helps you program graphics. The program consists of 23 machine-language routines that you access with the appropriate **POKE** and **USR** commands. For example, some of the programs are:

1. DRAW/UNDRAW—draws or erases multicharacter shapes with a single **POKE**. This feature allows you to add more realistic shapes, such as robots and spaceships, to your games, because machine-language drawing and undrawing is done quickly.
2. BORDER/UNBORDER—adds or erases a border to the TV screen to enhance the appearance of your display.
3. FILL—fills a specified area of the screen with the character you indicate.
4. REVERSE—changes the characters in a specified region of memory to their inverse video counterparts.
5. SEARCH & REPLACE—searches the screen for a particular character and replaces it with another that you specify.

Many other routines are also included with the Graphics Kit.

T/S Destroyer —2K

In this game, you attempt to destroy enemy ships before they can launch an attack on your planet. You must evade robot ships, guided missiles, and meteors. Safeguard the universe!

2K Games Pack —2K

This tape contains a number of game programs, including

1. Monster Masher
2. Star Blaster
3. Dragon Slayer
4. Copter Patrol

5. Killer Whale
6. Astro Walk

Alien Invasion —16K

Using bases, shields, and missiles, you must defeat the alien flying saucers in this game.

Mazogs —16K

You must find the treasure hidden within an intricate maze that is guarded by the terrible Mazogs. Your only assets are your wits, swords, and the help of the Mazogs' prisoners.

Space Command —16K

Prevent humanity from being enslaved by the evil androids of Zircon 12! Your friendly T/S 1000 will respond to short English commands as you defend humanity in this adventure game.

Red Alert —16K

Pilot your spaceship over treacherous mountains as you seek to destroy enemy fuel dumps with your missiles. But watch out for the enemy aliens seeking to destroy you.

Meteorites —16K

Use your spaceship to destroy the dangerous meteorites plaguing the space lanes. You have controls to rotate the ship, apply thrust, and fire missiles.

Shark's Treasure —16K

Your T/S 1000 will help you find the pirates' treasure, hidden in perilous shark-infested waters. Use short English commands to move around and manipulate objects in this adventure game.

Hangman —16K

This classic word game helps children learn how to spell. There are five, predefined libraries of words for the following programs:

1. Animals
2. Science
3. States
4. Countries
5. Fruits and Vegetables

The sixth program on this tape lets two people play against each other. The seventh program allows you to create and save your own library of words.

Biorhythms —16K

According to the Theory of Biorhythms, peoples' lives are strongly affected from birth by three cycles—emotional, physical, and intellectual—called biorhythm cycles, or simply biorhythms. This program plots your biorhythms to help you try to plan ahead.

The Stock Market Calculator —16K

This tape includes a number of programs to help you in the Stock Market, including:

1. NET PRICE WORKSHEET—calculates commissions on stocks and options. It also gives the total cost to buy or net from sale.
2. PROFIT/LOSS—calculates net dollars, commissions, and profit/loss based on stock or option sale prices.

and other programs.

The Financial Manager and Record Keeper —16K

This program helps you budget by letting you input and store income and expenditures for a year. Data is input by categories on a monthly basis, and the T/S 1000 keeps a running tabulation of the monthly income and expenses. The program can also project future expense versus income for financial planning.

Gladstone Electronics also offers a variety of software, including:

Toolkit — 16K

This program provides nine functions to help you write programs.

1. RENUM — renumbers program lines, including statements with GOTO, GOSUB, RUN, and LIST commands.
2. DEL — deletes a group of lines that you indicate by selecting the first and last line numbers.
3. MEM — shows you how much memory you have.
4. DUMP — prints a list of the current values of all string and numeric variables, except arrays and control variables.
5. FIND — finds and prints the line number of any string with up to 255 characters which you select.
6. REPLACE — replaces any string of up to 255 characters with a string that you select.
7. SAVE — stores the program in memory above RAMTOP. This function can be used in conjunction with APPEND.
8. APPEND — is used after you load a program into memory to append the program you saved above RAMTOP to the program currently in memory.
9. REMKILL — deletes all REM statements from a program.

80 Hours Around Europe — 16K

Try to win a fortune by visiting all the countries of Europe in 80 hours and obtaining currency from each country.

Galaxy Invaders — 16K

Defend your base against the alien spacecraft! You can move left and right while firing your laser against the menacing horde. But watch out because the pace of attack increases automatically as you score points.

ZX Galaxia — 16K

A fleet of ravenous Thargons is attacking the Earth. Can you foil their evil plans to conquer the world? Beware of their swift attack ships and deadly bombs!

ZX Chess I — 16K

Can you beat the computer at chess? You can select from six levels of difficulty.

Level 1 — The computer takes 10 seconds to reply to your move.

Level 2 — 30 to 40 seconds per move.

Level 3 — 3 to 8 minutes per move.

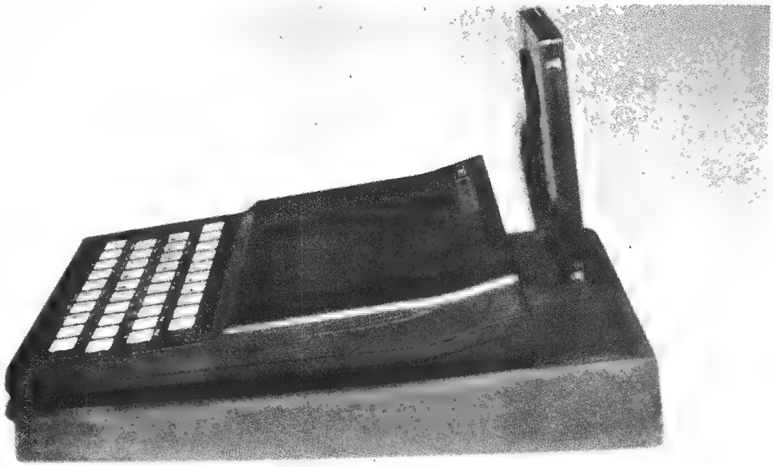
Levels 4, 5, and 6 are for extended or correspondence games.

You can save the game on tape at any point for continuation later. You can also print a copy of the board, print all moves up to a certain point, analyze the game by examining possible moves, swap places with the computer, and other actions.

Physical Enhancements

In addition to software and hardware, you can also obtain products to enhance the physical aspects of your T/S 1000. For example, you can get a Carrying and Storage Case for the T/S 1000 from Sight & Sound. This case is foam-lined with precut squares that let you customize the foam to fit your computer and anything else you want to put in the case, such as a tape recorder. Take your T/S 1000 along on your next vacation or business trip, and you will have really expanded its use beyond the horizon.

The Computer Pad from Zopf Industries is another aid for your T/S 1000. This pad (1) prevents the computer from sliding around, (2) tilts the T/S 1000 to make it more convenient for you to press keys, and (3) holds a cassette. If you'd like to raise your TV above the table or bench it sits on for easier viewing, the CRT Riser from Zopf is a platform that can serve this purpose.



Vendors of T/S 1000 Hardware and Software

Softsync, Incorporated
14 E. 34th Street
New York, NY 10016
(212) 685-2080

Gladstone Electronics
1585 Kenmore Avenue
Buffalo, NY 14217
(716) 874-5510

Memotech Corporation
7550 W. Yale Avenue
Denver, CO 80227
(303) 986-1516

Personal Software Services
482 Stoney Stanton Road
Coventry CV6 5FE
England
Coventry (0203) 667556

Bob Berch Software
19 Jacques Street
Rochester, NY 14620
(716) 244-0533

Sight & Sound
P.O. Box 29177
Richmond, VA 23229
(804) 741-0745

Zopf Industries
121 W. Mt. Hope
Lansing, MI 48910
(517) 487-9284

Magazines

SYNC
9 E. Hanover Avenue
Morris Plains, NJ 07950

Syntax
Bolton Road
R.D. 2 Box 457
Harvard, MA 0I45I

Byte
P.O. Box 328
Hancock, NH 03449

Compute!

P.O. Box 5406
Greensboro, NC 27403

APPENDIX A
Magazine Articles of Interest to T/S 1000 Users

Sync Magazine

GAMES AND PROGRAMS

Lunar Lander	Jan./Feb. 1982—Vol. 2, No. 1
Battleship Solitaire	Jan./Feb. 1982—Vol. 2, No. 1
Tioga Toads	Jan./Feb. 1982—Vol. 2, No. 1
Dice and Train	Jan./Feb. 1982—Vol. 2, No. 1
6 Shooter	Mar./Apr. 1982—Vol. 2, No. 2
Isolation	Mar./Apr. 1982—Vol. 2, No. 2
Space Warp	May/June 1982—Vol. 2, No. 3
ZX Destroyer	July/Aug. 1982—Vol. 2, No. 4
Galaxy Invaders	July/Aug. 1982—Vol. 2, No. 4
Micro Invaders	July/Aug. 1982—Vol. 2, No. 4
Comet Crusher	July/Aug. 1982—Vol. 2, No. 4
Crossing the Astroid Belt	July/Aug. 1982—Vol. 2, No. 4
Alien Treasure	July/Aug. 1982—Vol. 2, No. 4

MATH

Understanding Floating-Point Arithmetic Part One—Decimal and binary formats	Jan./Feb. 1982—Vol. 2, No. 1
Linear Regression	Jan./Feb. 1982—Vol. 2, No. 1
Understanding Floating-Point Arithmetic	

Part Two—The CALCULATOR Language	Mar./Apr. 1982—Vol. 2, No. 1
Plotting with 4K Basic	Mar./Apr. 1982—Vol. 2, No. 1
Least Squares Data Analysis with the ZX80/81	May/June 1982—Vol. 2, No. 3
Understanding Floating-Point Arithmetic Part Three	July/Aug. 1982—Vol. 2, No. 4

HARDWARE

Hardware Helpers	Jan./Feb. 1982—Vol. 2, No. 1
Getting Loaded	Mar./Apr. 1982—Vol. 2, No. 1
Two-Switch Human Interface for the Communicative Impaired	May/June 1982—Vol. 2, No. 3
Keyboard/System Conversion: The First 40/1 Keys	May/June 1982—Vol. 2, No. 3
Double Your Memory	July/Aug. 1982—Vol. 2, No. 4

MACHINE CODE

How to Invent a Game	Jan./Feb. 1982—Vol. 2, No. 1
The Game of Life Revisited—An Assembly Version	Jan./Feb. 1982—Vol. 2, No. 1
Window	Jan./Feb. 1982—Vol. 2, No. 1
LSCROLL—For Spectacular Screen Displays	Mar./Apr. 1982—Vol. 2, No. 1
How to Invent a Game	Mar./Apr. 1982—Vol. 2, No. 1
Safe Machine Code Routines	Mar./Apr. 1982—Vol. 2, No. 1

SYNTAX Magazine

PROGRAMS

Brownian Motion	Dec., 1982—Vol. 2, No. 12
Bytes Remaining	Dec., 1981—Vol. 2, No. 12
Convergence Pattern	Dec., 1981—Vol. 2, No. 12
Happy Holidays	Dec., 1981—Vol. 2, No. 12
Job Review	Dec., 1981—Vol. 2, No. 12
Slalom Course	Dec., 1981—Vol. 2, No. 12
Bar Chart	Jan., 1982—Vol. 3, No. 1
Breakeven Analysis	Jan., 1982—Vol. 3, No. 1
Binary Representation	Jan., 1982—Vol. 3, No. 1
Equational AND/OR	Jan., 1982—Vol. 3, No. 1
Simulate READ, DATA	Jan., 1982—Vol. 3, No. 1
Awari	Feb., 1982—Vol. 3, No. 2
Memory PEEK	Feb., 1982—Vol. 3, No. 2
Moving Billboard	Feb., 1982—Vol. 3, No. 2
Phone Number Word Generator	Feb., 1982—Vol. 3, No. 2
Salary Scheduling	Feb., 1982—Vol. 3, No. 2
Solving Simultaneous	Feb., 1982—Vol. 3, No. 2

Income Tax	Mar., 1982—Vol. 3, No. 3
Flashing Prompts	Mar., 1982—Vol. 3, No. 3
Video Artist	Mar., 1982—Vol. 3, No. 3
Dice Roll	Apr., 1982- Vol. 3, No. 4
Flower Plot	Apr., 1982—Vol. 3, No. 4
Side-Scrolling Billboard	Apr., 1982—Vol. 3, No. 4
34 Columns	Apr., 1982—Vol. 3, No. 4
READ, DATA RESTORE	Apr., 1982—Vol. 3, No. 4
Simulating Interger	Apr., 1982—Vol. 3, No. 4
Numbers Held Inexactly	May, 1982—Vol. 3, No. 5
Annuity	Jun., 1982—Vol. 3, No. 6
Line Renumbering	Jun., 1982—Vol. 3, No. 6
Loading Hex Listings	Jun., 1982—Vol. 3, No. 6
Speed Up PRINT Lines	Apr., 1982—Vol. 3, No. 4
Flying Bird	July, 1982—Vol. 3, No. 7
Grandson of Big Characters	July, 1982—Vol. 3, No. 7
Household Finances	July, 1982—Vol. 3, No. 7
Improved Data Input	July, 1982—Vol. 3, No. 7
Blast Effects	Aug., 1982—Vol. 3, No. 8
Cardioid Plot	Aug., 1982—Vol. 3, No. 8
Count	Aug., 1982—Vol. 3, No. 8
Diagonal Lines	Aug., 1982—Vol. 3, No. 8
Entrapment	Aug., 1982—Vol. 3, No. 8
MC Screen Fill	Aug., 1982—Vol. 3, No. 8
Sine Name	Aug., 1982—Vol. 3, No. 8
Shifting Maze	Aug., 1982—Vol. 3, No. 8
Spaces in REM	Aug., 1982—Vol. 3, No. 8
String Functions	Aug., 1982—Vol. 3, No. 8
Syntactic Sum Loader	Aug., 1982—Vol. 3, No. 8
Printing Strings Faster	Aug., 1982—Vol. 3, No. 8
Printing Bottom Line	Aug., 1982—Vol. 3, No. 8
Monthly Calendar	Sept., 1982—Vol. 3, No. 9
Significant Figures Subroutine	Sept., 1982—Vol. 3, No. 9
Math Function Execution	Sept., 1982—Vol. 3, No. 9
Maximize Display File for MC	Sept., 1982—Vol. 3, No. 9

HARDWARE PROJECTS

Video Monitor Driver	Dec., 1981—Vol. 2, No. 12
Loading Filter	Jan., 1982—Vol. 3, No. 1
ZX81 RAM Decoding	Feb., 1982—Vol. 3, No. 2
Memory Decoding	May, 1982—Vol. 3, No. 5
Problem-Solvers	May, 1982—Vol. 3, No. 5
2K Conversion	May, 1982—Vol. 3, No. 5
Adding a Joystick to ZX80/81	Jun., 1982—Vol. 3, No. 6
Regulator and Battery Backup	Aug., 1982—Vol. 3, No. 8

TIPS AND HINTS

Dim as Sum-to-Memory	Feb., 1982—Vol. 3, No. 2
Loading Technique	Apr., 1982—Vol. 3, No. 4
Hidden 8K Programs	Apr., 1982—Vol. 3, No. 4
Charger Ripple	Apr., 1982—Vol. 3, No. 4
Software Tips	Apr., 1982—Vol. 3, No. 4

MACHINE LANGUAGE

8K Routine Addresses	Jan., 1982—Vol. 3, No. 1
Binary Addition and Subtraction	Feb., 1982—Vol. 3, No. 2
Flags of 8K ROM	Feb., 1982—Vol. 3, No. 2
Flags Register	Mar., 1982—Vol. 3, No. 3
Numbers Held Inaccurately	Mar., 1982—Vol. 3, No. 3
Exchanging Registers	Jun., 1982—Vol. 3, No. 6
Keyboard Scanning	July, 1982—Vol. 3, No. 7
CP Branching Tests	Aug., 1982—Vol. 3, No. 8
Understanding the Stack	Aug., 1982—Vol. 3, No. 8
Testing Bits for Branching	Sept., 1982—Vol. 3, No. 9

BEGINNERS

Translating BASICs	Feb., 1982—Vol. 3, No. 2
Loading Registers	Apr., 1982—Vol. 3, No. 4
BASIC-Inkey$	May, 1982—Vol. 3, No. 5
ML-Register/Register	May, 1982—Vol. 3, No. 5
More on INKEY$	July, 1982—Vol. 3, No. 7
PUSHing and POPping	July, 1982—Vol. 3, No. 7

APPENDIX B
Order of Priority of Operators

Operator	Priority
Slicing and Subscripting	12
All Functions	11
**	1Ø
- (Unary Minus)	9
*,/	8
+,- (Binary Minus)	6
=,<=,>=,<,>,<>	5
NOT	4
AND	3
OR	2

Note: Operators of the same priority are evaluated left to right.

APPENDIX C

Reprinted from Timex *User Manual* by permission of Timex Computer Corporation.

Report Codes

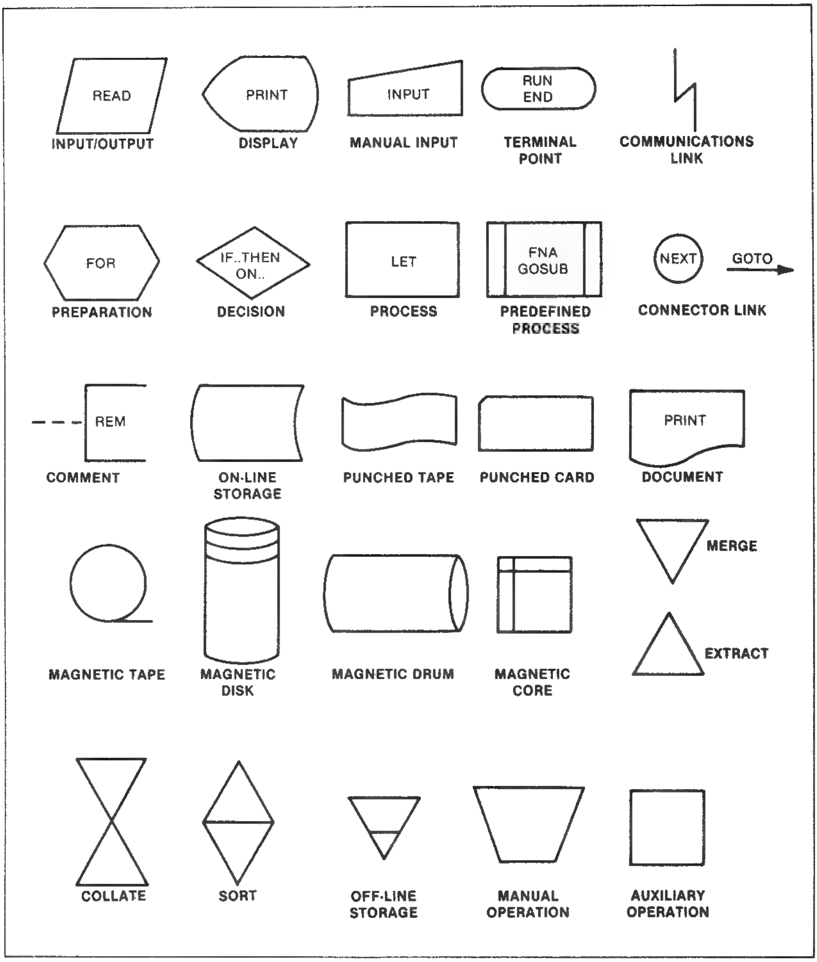
This table gives each report code, with a general description and a list of the statements and functions in which it can occur. In Chapter 21, under each statement or function, you will find a more detailed description of what the error reports mean.

Code	*Meaning*	*Situations*
Ø	Successful completion, or jump to line number bigger than any existing. A report with code Ø does not change the line number used by **CONT**.	Any
1	The control variable does not exist (has not been set up by a **FOR** statement), but there is an ordinary variable with the same name.	**NEXT**
2	An undefined variable has been used. For a simple variable this will happen if the variable is used before it has been assigned to in a **LET** statement. For a subscripted variable it will happen if the variable is used before it has been dimensioned in a **DIM** statement. For a control variable in a **FOR** statement and if there is no ordinary simple variable with the same name.	Any
3	Subscript out of range. If the subscript is out of range (negative, or bigger than 65535), error B will result.	Subscripted variables
4	Not enough room in memory. Note that the line number in the report (after the /) may not be complete on the screen,	**LET, INPUT, DIM, PRINT, LIST, PLOT, UNPLOT, FOR,**

Report Codes

Code	*Meaning*	*Situations*
	because of the shortage of memory: for instance, 4/2Ø may appear as 4/2. See Chapter 9.	**GOSUB**. Sometimes during function evaluation.
5	No more room on the screen. **CONT** will make room by clearing the screen.	**PRINT, LIST, PLOT, UNPLOT**
6	Arithmetic overflow: calculations have led to a number greater than about 1Ø^{38}.	Any arithmetic
7	No corresponding **GOSUB** for a **RETURN** statement.	**RETURN**
8	You have attempted an **INPUT** command (not allowed).	**INPUT**
9	**STOP** statement executed. **CONT** will not try to reexecute the **STOP** statement.	**STOP**
A	Invalid argument to certain functions.	**SQR, LN, ASN, ACS**
B	Integer out of range. When an integer is required, the floating-point argument is rounded to the nearest integer. If this is outside a suitable range, error B results.	**RUN, RAND, POKE, DIM, GOTO, GOSUB, LIST, PAUSE, PLOT, UNPLOT, CHR$, PEEK, USR**
	For array access, see also Report 3.	Array access
C	The text of the (string) argument of **VAL** does not form a valid numerical expression.	**VAL**
D	(i) Program interrupted by **BREAK**.	At the end of any statement, or in **LOAD, SAVE, LPRINT, LLIST,** or **COPY**.
	(ii) The **INPUT** line starts with **STOP**.	**INPUT**
E	Not used	
F	The program name provided is the empty string.	**SAVE**

APPENDIX D



The ANSI Flow Chart Symbols

APPENDIX E
Derivation of the Periodic Payment Formula

Generally, the payment of a loan in equal installments includes portions towards both the interest and the principal.

the first month,

(1) $P=A*I+T1$

in which

P = periodic payment

A = amount borrowed

I = decimal interest per period

For example, if the yearly interest is 12%, and the loan is paid monthly, then

$$I = \frac{12}{100*12} = .01$$

T1 = portion of payment that applies toward reducing the amount borrowed in the first month.

A*I = is the portion of the payment that applies toward the interest.

In the second month,

$$(2)\ P=(A-T1)*I+T2$$

in which the amount owed is reduced by P1, the portion of the first payment that applied toward the amount borrowed in the first month.

Equations I and 2 can be set equal to one another, since they both have the same payment, P, as in

$$\begin{aligned} A*I+T1 &= (A-T1)*I+T2 \\ &= A*I-T1*I+T2 \end{aligned}$$

To solve for T2 in terms of T1, use

$$T2=T1*(1+I)$$

In the third month,

$$(3)\ P=(A-T1-T2)*I+T3$$

Setting equation 2 equal to equation 3, and multiplying through by I, gives

$$A*I-T1*I+T2=A*I-T1*I-T2*I+T3$$

then

$$T2=-T2*I+T3$$

and so

$$T3=T2*(1+I)=T1*(1+I)**2$$

Now we can deduce that the last payment applying toward the amount borrowed can be expressed as

$$TN=T1*(1+I)**(N-1)$$

in which N is the total number of payments. Also, the last payment is

$$P=(A-T1-T2-T3...-(TN-1))*I+TN$$
$$P=TN*I+TN= TN*(1+I)$$

since we must pay off the loan by TN in the last payment. So

$$P=(T1*(1+I)**N-1)*(1+I)$$
$$P=T1*(1+I)**N$$
$$P=(P-A*I)*(1+I)**N$$

Solving this last equation for P gives

$$P=\frac{A*I*(1+I)**N}{(1+I)**N-1}$$

APPENDIX F
BASIC Symbols and Keys

Symbol	Key	Position
ABS	G	Below key
AND	2	Above key
ARCCOS	S	Below key
ARCSIN	A	Below key
ARCTAN	D	Below key
BREAK	SPACE	Above key
CHR$	U	Below key
CLEAR	X	Above key
CLS	V	Above key
CODE	I	Below key
CONT	C	Above key
COPY	Z	Above key
COS	W	Below key
DELETE	Ø	Above key
DIM	D	Above key
EDIT	1	On key
EXP	X	Below key
FAST	F	On key
FOR	F	Above key
FUNCTION	ENTER	On key

GOSUB	H	Above key
GOTO	G	Above key
IF	U	Above key
INKEY$	B	Below key
INPUT	I	Above key
INT	R	Below key
LEN	K	Below key
LET	L	Above key
LIST	K	Above key
LLIST	G	On key
LN	Z	Below key
LOAD	J	Above key
LPRINT	S	On key
NEW	A	Above key
NEXT	N	Above key
NOT	N	Below key
OR	W	On key
PAUSE	M	Above key
PEEK	O	Below key
PI	M	Below key
PLOT	Q	Above key
POKE	O	Above key
PRINT	P	Above key
RAND	T	Above key
REM	E	Above key
RETURN	Y	Above key
RND	T	Below key
RUN	R	Above key
SAVE	S	Above key
SGN	F	Below key
SIN	Q	Below key
SLOW	D	On key
SQR	H	Below key
STEP	E	On key
STOP	A	On key
STR$	Y	Below key
TAB	P	Below key
TAN	E	Below key
THEN	3	On key

TO	4	On key
UNPLOT	W	Above key
USR	L	Below key
VAL	J	Below key

APPENDIX G
BASIC Functions and Statements

Reprinted from Timex *User Manual* by permission of Timex Computer Corporation

Function	Type of operand	Result
	(x)	
ABS	number	Absolute magnitude.
ACS	number	Arccosine in radians. Error A if x not in the range −1 to +1.
AND	binary operation, right operand always a number.	
	Numeric left operand:	A **AND** B = $\left\{\begin{array}{l} \text{A if B} \neq Ø \\ Ø \text{ if B} = Ø \end{array}\right.$
	String left operand:	A$ **AND** B = $\left\{\begin{array}{l} \text{A\$ if B} \neq Ø \\ \text{"" if B} = Ø \end{array}\right.$
ASN	number	Arcsine in radians. Error A if x not in the range −1 to +1.

ATN	number	Arctangent in radians.
CHR$	number	The character whose code is x, rounded down to the nearest integer. Error B if x not in the range Ø to 255.
CODE	string	The code of the first character in x (or Ø if x is empty string).
COS	number (in radians)	Cosine
EXP	number	e^x.
INKEY$	none	Reads the keyboard. The result is the character representing (in L mode) the key pressed if there is exactly one, else the empty string.
INT	number	Integer part (always rounds down).
LEN	string	Length.
LN	number	Natural logarithm (to base e). Error A if x < = Ø.
NOT	number	Ø if x ≠ Ø, 1 if x = Ø. **NOT** has priority 4.
OR	binary operation, both operands numbers	$A \text{ } \mathbf{OR} \text{ } B = \begin{cases} 1 \text{ if } B \neq \text{Ø}. \\ A \text{ if } B = \text{Ø}. \end{cases}$ **OR** has priority 2.
PEEK	number	The value of the byte in memory whose address is x (rounded to the nearest integer). Error B if x not in the range Ø to 65535.
PI	none	π (3.14159265 . . .)
RND	none	The next pseudo-random number y in a sequence generated by taking the powers of 75 modulo 65537, subtracting 1 and dividing by 65536. Ø < = y < 1.
SGN	number	Signum: the sign (−1, Ø or +1) of x.

SIN	number (in radians)	Sine.
SQR	number	Square root. Error B if x < Ø.
STR$	number	The string of characters that would be displayed if x were printed.
TAN	number (in radians)	Tangent.
USR	number	Calls the machine-code subroutine whose starting address is x. On return, the result is the contents of the bc register pair.
VAL	string	Evaluates x (without its bounding quotes) as a numerical expression. Error C if x contains a syntax error, or gives a string value. Other errors possible, depending on the expression.
–	number	Negation

The following are binary operations:

+	Addition (on numbers), or concatenation (on strings)	
–	Subtraction	
*	Multiplication	
/	Division	
**	Raising to a power. Error B if left operand is negative.	
=	Equals	Both operands must be of the same type. The result is a number, 1 if the comparison holds, and Ø if it does not.
>	Greater than	
<	Less than	
<=	Less than or equal to	
>=	Greater than or equal to	
<>	Not equal to	

Statements

In this list,

@ represents a single letter

v represents a variable

x,y,z represents numerical expressions

m,n represents numerical expressions that are rounded to the nearest integer

e	represents an expression
f	represents a string-valued expression
s	represents a statement

Note that arbitrary expressions are allowed everywhere (except for the line number at the beginning of a statement).

All statements except **INPUT** can be used either as commands or in programs (although they may be more sensible in one than the other).

CLEAR

Deletes all variables, freeing the space they occupied.

CLS

(CLear Screen) Clears the display file. See Chapter 26 concerning the display file.

CONT

Suppose a/b were the last report with a non-zero. Then **CONT** has the effect

GOTO b if a ≠ 9
GOTO b+1 if a = 9 (**STOP** statement)

COPY

Sends a copy of the display to the printer, if attached; otherwise does nothing.
Report D if **BREAK** pressed.

DIM @ ($n_1,...,n_k$)

Deletes any array with the name @ and sets up an array of numbers with k dimensions $n_1,...,n_k$. Initializes all the values to Ø.
Error 4 occurs if there is no room to fit the array in. An array is undefined until it is dimensioned in a **DIM** statement.

DIM @ $ ($n_1, ..., n_k$)

Deletes any array or string with the name @ $ and sets up an array of characters with k dimensions $n_1, ..., n_k$. Initializes all the values to "". This can be considered as an array of strings of fixed length n_k, with k−1 dimensions $n_1, ..., n_{k-1}$.
Error 4 occurs if there is no room to fit the array in. An array is undefined until it is dimensioned in a **DIM** statement.

FAST

Starts fast mode, in which the display file is displayed only at the end of the program, while **INPUT** data is being typed in, or during a pause.

FOR@ =x **TO** y

FOR@ =x **TO** y **STEP** 1

FOR@ =x **TO** y **STEP** z

Deletes any simple variable and sets up a control variable with value x, limit y, step z, and looping address 1 more than the line number of the **FOR** statement (−1 if it is a command). Checks if the initial value is greater (if step >= Ø) or less (if step < Ø) than the limit, and if so, then skips to statement **NEXT**@ at the beginning of a line. See **NEXT**@.
Error 4 occurs if there is no room for the control variable.

GOSUB n

Pushes the number of the **GOSUB** statement onto a stack; then as **GOTO** n.
Error 4 can occur if there are not enough **RETURN**s.

GOTO n

Jumps to line n (or, if there is none, to the first line after that).

IF x **THEN** s

If x is true (non-zero), then s is executed.
The form '**IF** x **THEN** line number' is not allowed.

INPUT v

Stops (with no special prompt) and waits for the user to type in an expression; the value of this is assigned to v. In fast mode, the display file is displayed. **INPUT** cannot be used as a command; error 8 occurs if you try.
If the first character in the **INPUT** line is **STOP**, the program stops with report D.

LET v=e

Assigns the value of e to the variable v.
LET cannot be omitted.
A simple variable is undefined until it is assigned to in a **LET** or **INPUT** statement.
If v is a subscripted string variable, or a sliced string variable (substring), then the assignment is *Procrustean*: the string value of e is either truncated or filled out with spaces to the right, to make it the same length as the variable v.

LIST

LIST Ø

LIST n

Lists the program on the TV screen, starting at line n, and makes n the current line.
Error 4 or 5 if the listing is too long to fit on the screen; **CONT** will do exactly the same again.

LLIST

LLIST Ø

LLIST n

Like **LIST**, but using the printer instead of the television.
Should do nothing if the printer is not attached.
Stops with Report D if **BREAK** is pressed.

LOAD f

Looks for a program called f on tape and loads it and its variables. If f = "", then loads the first program available.
If **BREAK** is pressed, then
(i) if no program has yet been read in from tape, stops with report D and old program;
(ii) if part of a program has been read in, then executes **NEW**.

LPRINT ...

Like **PRINT**, but using the printer instead of the television. A line of text is sent to the printer
(i) when printing spills over from one line to the next,
(ii) after an **LPRINT** statement that does not end in a comma or a semicolon,
(iii) when a comma or **TAB** item requires a new line, or
(iv) at the end of the program, if there is anything left unprinted.
In an **AT** item, only the column number has any affect; the line number is ignored. An **AT** item never sends a line of text to the printer.
There should be no effect if the printer is absent.
Stops with report D if **BREAK** is pressed.

NEW

Restarts the BASIC system, deleting program and variables and using the memory up to but not including the byte whose address is in the system variable RAMTOP (bytes 16388 and 16389).

NEXT@

(i) Finds the control variable @.
(ii) Adds its step to its value.
(iii) If the step >= Ø and the value > the limit; or if the step < Ø and the value < the limit, then jumps to the looping line.
Error 1 if there is no control variable @.

PAUSE n

Stops computing and displays the display file for n frames (at 5Ø frames per second) or until a key is pressed. Ø <= n <= 65535, else error B. If n >= 32767, then the pause is not timed, but lasts until a key is pressed.

PLOT m,n

Blacks in the pixel (|m|,|n|); moves the **PRINT** position to just after that pixel.
Ø <= |m| <= 63, Ø <= |n| <= 43, else error B.

POKE m,n

Writes the value n to the byte in store with address m.
Ø <= m <= 65535, −255 <= n <= 255, else error B.

PRINT ...

The '...' is a sequence of **PRINT** items, separated by commas or semicolons. They are written to the display file for display on the television. The position (line and column) where the next character is to be printed is called the **PRINT** position.
A **PRINT** item can be

(i) empty, i.e., nothing
(ii) a numerical expression

First, a minus sign is printed if the value is negative.
Now let x be the modulus of the value.

If x <= $1Ø^{-5}$ or x >= $1Ø^{13}$, then it is printed using scientific notation. The mantissa part has up to eight digits (with no trailing zeros), and the decimal point (absent if only one digit) is after the first. The exponent part is E, followed by + or −, followed by one or two digits.

Otherwise x is printed in ordinary decimal notation with up to eight significant digits, and no trailing zeros after the decimal point. A decimal point right at the beginning is always followed by a zero, so, for instance, .Ø3 and Ø.3 are printed as such.

Ø is printed as a single digit Ø.

(iii) a string expression.

The tokens in the string are expanded, possibly with a space before or after.

The quote image character prints as ". Unused characters and control characters print as ?.

(iv) **AT** m,n

The **PRINT** position is changed to line m (counting from the top), column n (counting from the left). Ø < =|m|< = 21, Ø < =|n|< = 31, else error B. If |m| = 22 or 23, error 5.

(v) **TAB** n

n is reduced modulo 32. Then, the **PRINT** position is moved to column n, staying on the same line unless this would involve backspacing, in which case it moves on to the next line. Ø < = n < = 255, else error B.

A semicolon between two items leaves the **PRINT** position unchanged, so that the second item follows immediately after the first. A comma, on the other hand, moves the **PRINT** position on at least one place; and after that, as many as are necessary to leave it in column Ø or 16, moving to a new line if necessary.

At the end of the **PRINT** statement, if it does not end in a semicolon or comma, a new line is started.

Error 4 (out of memory) can occur with 3K or less of memory.
Error 5 means that the screen is filled.

In both cases, the cure is **CONT**, which will clear the screen and allow the program to continue.

RAND

RAND Ø

RAND n

Sets the system variable (called SEED) used to generate the next value of **RND**. If n ≠ Ø, the SEED is given the value n; if n = Ø, it is given the value of another system variable (called FRAMES) that counts the frames so far displayed on the television, and so should be fairly random.
Error B occurs if n is not in the range Ø to 65535.

REM ...

No effect. '...' can be any sequence of characters except **ENTER**.

RETURN

Pops a line number from the **GOSUB** stack and jumps to the line after it.

Error 7 occurs when there is no line number on the stack. There is some mistake in your program; **GOSUB**s are not properly balanced by **RETURN**s.

RUN

RUN Ø

RUN n

CLEAR, and then **GOTO** n.

SAVE f

Records the program and variables on tape and calls it f.

SAVE should not be used inside a **GOSUB** routine.

Error F occurs if f is the empty string, which is not allowed.

SCROLL

Scrolls the display file up one line, losing the top line and making an empty line at the bottom.

Note that the new line is genuinely empty with just an **ENTER** character and no spaces.

SLOW

Puts the computer into compute and display mode, in which the display file is displayed continuously and computing is done during the spaces at the top and bottom of the picture.

STOP

Stops the program with Report 9. **CONT** will resume with the following line.

UNPLOT m,n

Like **PLOT**, but blanks out a pixel instead of blacking it in.

APPENDIX H
The System Variables

Reprinted from Timex *User Manual* by permission of Timex Computer Corporation

The System Variables

The bytes in memory from 16384 to 165Ø8 are set aside for specific uses by the system. You can peek them to find out various things about the system, and some of them can be usefully poked. They are listed here with their uses.

These are called system variables and carry names, but do not confuse them with the variables used by the BASIC. You cannot use the names in a BASIC program; they are simply mnemonics that are used to make it easier to refer to the variables.

The abbreviations in column 1 have the following meanings.

X	The variable should not be poked, because the system might crash.
N	Poking the variable will have no lasting affect.
S	The variable is saved by **SAVE**.

The number in column 1 is the number of bytes in the variable. For two bytes, the first one is the *less* significant byte — the reverse of what you might expect. So to poke a value v to a two-byte variable at address n, use

```
POKE n,v−256*INT (v/256)
POKE n+1,INT (v/256)
```

and to peek its value, use the expression:

```
PEEK n + 256*PEEK (n+1)
```

Notes	*Address*	*Name*	*Contents*
1	16384	ERR_NR	1 less than the report code. Starts off at 255 (for −1), so **PEEK** 16384, if it works at all, gives 255. **POKE** 16384, n can be used to force an error halt: Ø <= n <= 14 gives one of the usual reports, 15 <= n <= 34 or 99 <= n <= 127 gives a nonstandard report, and 35 <= n <= 98 is likely to mess up the display file.
X1	16385	FLAGS	Various flags to control the BASIC system.
X2	16386	ERR_SP	Address of first item on machine stack (after **GOSUB** returns).
2	16388	RAMTOP	Address of first byte above BASIC system area. You can poke this to make **NEW** reserve space above that area (see Chapter 25) or to fool **CLS** into setting up a minimal display file (Chapter 26).
N1	1639Ø	MODE	Specifies K, L, F or G cursor
N2	16391	PPC	Line number of statement currently being executed. Poking this has no lasting effect except in the last line of the program.
S1	16393	VERSN	Ø Identifies 8K ROM in saved programs.
S2	16394	E_PPC	Number of current line (with program cursor).
SX2	16396	D_FILE	See Chapter 26.
S2	16398	DF_CC	Address of **PRINT** position in display file. Can be poked so that **PRINT** output is sent elsewhere.
SX2	164ØØ	VARS	See Chapter 26.
SN2	164Ø2	DEST	Address of variable in assignment.
SX2	164Ø4	E_LINE	See Chapter 26.
SX2	164Ø6	CH_ADD	Address of the next character to be interpreted: the character after the argument of **PEEK**, or the **ENTER** at the end of a **POKE** statement.
S2	164Ø8	X_PTR	Address of the character preceding the marker.

Notes	*Address*	*Name*	*Contents*
SX2	1641Ø	STKBOT	See Chapter 26.
SX2	16412	STKEND	
SN1	16414	BREG	Calculator's b register.
SN2	16415	MEM	Address of area used for calculator's memory. (Usually MEMOT, but not always.)
S1	16417	not used	
SX1	16418	DF_SZ	The number of lines (including one blank line) in the lower part of the screen.
S2	16419	S_TOP	The number of the top program line in automatic listings.
SN2	16421	LAST_K	Shows which keys pressed
SN1	16423		Debounce status of keyboard.
SN1	16424	MARGIN	Number of blank lines above or below picture — 31.
SX2	16425	NXTLIN	Address of next program line to be executed.
S2	16427	OLDPPC	Line number to which **CONT** jumps.
SN1	16429	FLAGX	Various flags.
SN2	1643Ø	STRLEN	Length of string type designation in assignment.
SN2	16432	T-ADDR	Address of next item in syntax table (very unlikely to be useful).
S2	16434	SEED	The seed for **RND**. This is the variable that is set by **RAND**.
S2	16436	FRAMES	Counts the frames displayed on the television. Bit 15 is 1. Bits Ø to 14 are decremented for each frame sent to the television. This can be used for timing, but **PAUSE** also uses it. **PAUSE** resets bit 15 to Ø and puts in bits Ø to 14 the length of the pause. When these have been counted down to zero, the pause stops. If the pause stops because of a key depression, bit 15 is set to one again.
S1	16438	COORDS	x-coordinate of last point **PLOT**ted.
S1	16439		y-coordinate of the last point **PLOT**ted.
S1	1644Ø	PR_CC	Less significant byte of address of next position for **LPRINT** to print at (in PRBUFF).
SX1	16441	S_POSN	Column number for **PRINT** position.
SX1	16442		Line number for **PRINT** position.
S1	16443	CDFLAG	Various flags. Bit 7 is on (1) during compute and display e.

Notes	*Address*	*Name*	*Contents*
S33	16444	PRBUFF	Printer buffer (33rd character) is **ENTER**.
SN3Ø	16477	MEMBOT	Calculator's memory area; used to store numbers that cannot conveniently be put on the calculator stack.
S2	165Ø7	not used	

Exercises

1. Try this program

```
1Ø FOR N=Ø TO 21
2Ø PRINT PEEK (PEEK 164ØØ+256* PEEK 164Ø1 + N)
3Ø NEXT N
```

This tells you the first 22 bytes of the variables area: try to match up the control variable N with the description in Chapter 26.

2. In the program above, change line 2Ø to

```
2Ø PRINT PEEK (165Ø9+N)
```

This tells you the first 22 bytes of the program area. Match these up with the program itself.

APPENDIX I

Reprinted from Timex *User Manual* by permission of Timex Computer Corporation.

The Character Set

This is the complete T/S 1ØØØ character set, with codes in decimal and hex. If one imagines the codes being Z8Ø machine code instructions, then the right-hand columns give the corresponding assembly language mnemonics. As you are probably aware, certain Z8Ø instructions are compounds starting with CBh or EDh, as shown in the right-hand columns.

Code	*Character*	*Hex*	*Z8Ø assembler*	*-after CB*	*-after ED*
Ø	space	ØØ	nop	ric b	
1	▘	Ø1	ld bc,NN	ric c	
2	▝	Ø2	ld (bc), a	ric d	
3	▀	Ø3	inc bc	ric e	
4	▖	Ø4	inc b	ric h	
5	▌	Ø5	dec b	ric l	
6	▞	Ø6	ld b,N	ric (hl)	
7	▛	Ø7	rlca	ric a	
8	▒	Ø8	ex af,af'	rrc b	
9	🮏	Ø9	add hl,bc	rrc c	
1Ø	🮎	ØA	ld a,(bc)	rrc d	
11	"	ØB	dec bc	rrc e	
12	£	ØC	inc c	rrc h	
13	$	ØD	dec c	rrc l	
14	:	ØE	ld c,N	rrc (hl)	
15	?	ØF	rrca	rrc a	
16	(	1Ø	djnz DIS	rl b	

Code	*Character*	*Hex*	*Z8Ø assembler*	*-after CB*	*-after ED*
17	)	11	ld de,NN	rl c	
18	>	12	ld (de),a	rl d	
19	<	13	inc de	rl e	
2Ø	=	14	inc d	rl h	
21	+	15	dec d	rl l	
22	–	16	id d,N	rl (hl)	
23	*	17	rla	rl a	
24	/	18	jr DIS	rr b	
25	;	19	add hl,de	rr c	
26	,	1A	ld a,(de)	rr d	
27	.	1B	dec de	rr e	
28	Ø	1C	inc e	rr h	
29	1	1D	dec e	rr l	
3Ø	2	1E	ld e,N	rr (hl)	
31	3	1F	rra	rr a	
32	4	2Ø	jr nz,DIS	sla b	
33	5	21	ld hl,NN	sla c	
34	6	22	id (NN),hl	sla d	
35	7	23	inc hl	sla e	
36	8	24	inc h	sla h	
37	9	25	dec h	sla l	
38	A	26	ld h,N	sla (hl)	
39	B	27	daa	sla a	
4Ø	C	28	jr z,DIS	sra b	
41	D	29	add hl,hl	sra c	
42	E	2A	ld hl,(NN)	sra d	
43	F	2B	dec hl	sra e	
44	G	2C	inc l	sra h	
45	H	2D	dec l	sra l	
46	I	2E	ld l,N	sra (hl)	
47	J	2F	cpl	sra a	
48	K	3Ø	jr nc,DIS		
49	L	31	ld sp,NN		
5Ø	M	32	id (NN),a		
51	N	33	inc sp		
52	O	34	inc (hl)		
53	P	35	dec (hl)		
54	Q	36	id (hl),N		
55	R	37	scf		
56	S	38	jr c,DIS	srl b	
57	T	39	add hl,sp	srl c	
58	U	3A	ld a,(NN)	srl d	
59	V	3B	dec sp	srl e	
6Ø	W	3C	inc a	srl h	
61	X	3D	dec a	srl l	
62	Y	3E	ld a,N	srl (hl)	
63	Z	3F	ccf	srl a	

Code	*Character*	*Hex*	*Z8Ø assembler*	*-after CB*	*-after ED*
64	**RND**	4Ø	ld b,b	bit Ø,b	in b,(c)
65	**INKEY$**	41	ld b,c	bit Ø,c	out (c),b
66	**PI**	42	ld b,d	bit Ø,d	sbc hl,bc
67	not used	43	ld b,e	bit Ø,e	ld (NN),bc
68		44	ld b,h	bit Ø,h	neg
69		45	ld b,l	bit Ø,l	retn
7Ø		46	ld b,(hl)	bit Ø,(hl)	im Ø
71		47	ld b,a	bit Ø,a	ld i,a
72		48	ld c,b	bit 1,b	in c,(c)
73		49	ld c,c	bit 1,c	out (c),c
74		4A	ld c,d	bit 1,d	adc hl,bc
75		4B	ld c,e	bit 1,e	ld bc,(NN)
76		4C	ld c,h	bit 1,h	
77		4D	ld c,l	bit 1,l	reti
78		4E	ld c,(hl)	bit 1,(hl)	
79		4F	ld c,a	bit 1,a	ld r,a
8Ø		5Ø	ld d,b	bit 2,b	in d,(c)
81		51	ld d,c	bit 2,c	out (c),d
82		52	ld d,d	bit 2,d	sbc hl,de
83		53	ld d,e	bit 2,e	ld (NN),de
84		54	ld d,h	bit 2,h	
85		55	ld d,l	bit 2,l	
86		56	ld d,(hl)	bit 2,(hl)	im 1
87		57	ld d,a	bit 2,a	ld a,i
88		58	ld e,b	bit 3,b	in e,(c)
89		59	ld e,c	bit 3,c	out (c),e
9Ø		5A	ld e,d	bit 3,d	adc hl,de
91		5B	ld e,e	bit 3.e	ld de,(NN)
92		5C	ld e,h	bit 3,h	
93		5D	ld e,l	bit 3,l	
94		5E	ld e,(hl)	bit 3,(hl)	im 2
95		5F	ld e,a	bit 3,a	ld a,r
96		6Ø	ld h,b	bit 4,b	in h,(c)
97	not used	61	ld h,c	bit 4,c	out (c),h
98		62	ld h,d	bit 4,d	sbc hl,hl
99		63	ld h,e	bit 4,e	ld (NN),hl
1ØØ		64	ld h,h	bit 4,h	
1Ø1		65	ld h,l	bit 4,l	
1Ø2		66	ld h,(hl)	bit 4,(hl)	
1Ø3		67	ld h,a	bit 4,a	rrd
1Ø4		68	ld l,b	bit 5,b	in l,(c)
1Ø5		69	ld l,c	bit 5,c	out (c),l
1Ø6		6A	ld l,d	bit 5,d	adc hl,hl
1Ø7		6B	ld l,e	bit 5,e	ld de,(NN)
1Ø8		6C	ld l,h	bit 5,h	
1Ø9		6D	ld l,l	bit 5,l	
11Ø		6E	ld l,(hl)	bit 5,(hl)	
111		6F	ld l,a	bit 5,a	rid

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
112	cursor up ⇧	70	ld (hl),b	bit 6,b	
113	cursor down ⇩	71	ld (hl),c	bit 6,c	
114	cursor left ⇦	72	ld (hl),d	bit 6,d	sbc hl,sp
115	cursor right ⇨	73	ld (hl),e	bit 6,e	ld (NN),sp
116	**GRAPHICS**	74	ld (hl),h	bit 6,h	
117	**EDIT**	75	ld (hl),l	bit 6,l	
118	**ENTER**	76	halt	bit 6, (hl)	
119	**DELETE**	77	ld (hl),a	bit 6,a	
120	🅺 / 🅻 mode	78	ld a,b	bit 7,b	in a,(c)
121	**FUNCTION**	79	ld a,c	bit 7,c	out (c),a
122	not used	7A	ld a,d	bit 7,d	adc hl,sp
123	not used	7B	ld a,e	bit 7,e	ld sp,(NN)
124	not used	7C	ld a,h	bit 7,h	
125	not used	7D	ld a,l	bit 7,l	
126	number	7E	ld a,(hl)	bit 7,(hl)	
127	cursor	7F	ld a,a	bit 7,a	
128	█	80	add a,b	res 0,b	
129	▟	81	add a,c	res 0,c	
130	▙	82	add a,d	res 0,d	
131	▄	83	add a,e	res 0,e	
132	▜	84	add a,h	res 0,h	
133	▐	85	add a,l	res0,l	
134	▚	86	add a,(hl)	res 0,(hl)	
135	▗	87	add a,a	res 0,a	
136	▒	88	adc a,b	res 1,b	
137	🮑	89	adc a,c	res 1,c	
138	🮒	8A	adc a,d	res 1,d	
139	inverse "	8B	adc a,e	res 1,e	
140	inverse	8C	adc a,h	res 1,h	
141	inverse $	8D	adc a,l	res 1,l	
142	inverse :	8E	adc a,(hl)	res 1,(hl)	
143	inverse ?	8F	adc a,a	res 1,a	
144	inverse (	90	sub b	res 2,b	
145	inverse)	91	sub c	res 2,c	
146	inverse >	92	sub d	res 2,d	
147	inverse <	93	sub e	res 2,e	
148	inverse =	94	sub h	res 2,h	
149	inverse +	95	sub l	res 2,l	
150	inverse −	96	sub (hl)	res 2,(hl)	
151	inverse *	97	sub a	res 2,a	
152	inverse /	98	sbc a,b	res 3,b	
153	inverse ;	99	sbc a,c	res 3,c	
154	inverse ,	9A	sbc a,d	res 3,d	
155	inverse .	9B	sbc a,e	res 3,e	
156	inverse 0	9C	sbc a,h	res 3,h	
157	inverse 1	9D	sbc a,l	res 3,l	
158	inverse 2	9E	sbc a,(hl)	res 3,(hl)	
159	inverse 3	9F	sbc a,a	res 3,a	

Code	*Character*	*Hex*	*Z8Ø assembler*	*-after CB*	*-after ED*
16Ø	inverse 4	AØ	and b	res 4,b	ldi
161	inverse 5	A1	and c	res 4,c	cpi
162	inverse 6	A2	and d	res 4,d	ini
163	inverse 7	A3	and e	res 4,e	outi
164	inverse 8	A4	and h	res 4,h	
165	inverse 9	A5	and l	res 4,l	
166	inverse A	A6	and (hl)	res 4,(hl)	
167	inverse B	A7	and a	res 4,a	
168	inverse C	A8	xor b	res 5,b	ldd
169	inverse D	A9	xor c	res 5,c	cpd
17Ø	inverse E	AA	xor d	res 5,d	ind
171	inverse F	AB	xor e	res 5,e	outd
172	inverse G	AC	xor h	res 5,h	
173	inverse H	AD	xor l	res 5,l	
174	inverse I	AE	xor (hl)	res 5,(hl)	
175	inverse J	AF	xor a	res 5,a	
176	inverse K	BØ	or b	res 6,b	ldir
177	inverse L	B1	or c	res 6,c	cpir
178	inverse M	B2	or d	res 6,d	inir
179	inverse N	B3	or e	res 6,e	otir
18Ø	inverse O	B4	or h	res 6,h	
181	inverse P	B5	or l	res 6,l	
182	inverse Q	B6	or (hl)	res 6,(hl)	
183	inverse R	B7	or a	res 6,a	
184	inverse S	B8	cp b	res 7,b	lddr
185	inverse T	B9	cp c	res 7,c	cpdr
186	inverse U	BA	cp d	res 7,d	indr
187	inverse V	BB	cp e	res 7,e	otdr
188	inverse W	BC	cp h	res 7,h	
189	inverse X	BD	cp l	res 7,l	
19Ø	inverse Y	BE	cp (hl)	res 7,(hl)	
191	inverse Z	BF	cp a	res 7,a	
192	""""	CØ	ret nz	set Ø,b	
193	**AT**	C1	pop bc	set Ø,c	
194	**TAB**	C2	jp nz,NN	set Ø,d	
195	not used	C3	jpNN	set Ø,e	
196	**CODE**	C4	call nz,NN	set Ø,h	
197	**VAL**	C5	push bc	set Ø,l	
198	**LEN**	C6	add a,N	set Ø,(hl)	
199	**SIN**	C7	rst Ø	set Ø,a	
2ØØ	**COS**	C8	ret z	set 1,b	
2Ø1	**TAN**	C9	ret	set 1,c	
2Ø2	**ASN**	CA	jp z,NN	set 1,d	
2Ø3	**ACS**	CB		set 1,e	
2Ø4	**ATN**	CC	call z,NN	set 1,h	
2Ø5	**LN**	CD	call NN	set 1,l	
2Ø6	**EXP**	CE	adc a,N	set 1,(hl)	
2Ø7	**INT**	CF	rst 8	set 1,a	

Code	*Character*	*Hex*	*Z8Ø assembler*	*-after CB*	*-after ED*
2Ø8	**SQR**	DØ	ret nc	set 2,b	
2Ø9	**SGN**	D1	pop de	set 2,c	
21Ø	**ABS**	D2	jp nc,NN	set 2,d	
211	**PEEK**	D3	out N,a	set 2,e	
212	**USR**	D4	call nc,NN	set 2,h	
213	**STR$**	D5	push de	set 2,l	
214	**CHR$**	D6	sub N	set 2,(hl)	
215	**NOT**	D7	rst 16	set 2,a	
216	**	D8	ret c	set 3,b	
217	**OR**	D9	exx	set 3,c	
218	**AND**	DA	jp c,NN	set 3,d	
219	<=	DB	in a,N	set 3,e	
22Ø	>=	DC	call c,NN	set 3,h	
221	<>	DD	prefixes instructions using ix	set 3,l	
222	**THEN**	DE	sbc a,N	set 3,(hl)	
223	**TO**	DF	rst 24	set 3,a	
224	**STEP**	EØ	ret po	set 4,b	
225	**LPRINT**	E1	pop hl	set 4,c	
226	**LLIST**	E2	jp po,NN	set 4,d	
227	**STOP**	E3	ex (sp),hl	set 4,e	
228	**SLOW**	E4	call po,NN	set 4,h	
229	**FAST**	E5	push hl	set 4,l	
23Ø	**NEW**	E6	and N	set 4,(hl)	
231	**SCROLL**	E7	rst 32	set 4,a	
232	**CONT**	E8	ret pe	set 5,b	
233	**DIM**	E9	jp (hl)	set 5,c	
234	**REM**	EA	jp pe,NN	set 5,d	
235	**FOR**	EB	ex de,hl	set 5,e	
236	**GOTO**	EC	call pe,NN	set 5,h	
237	**GOSUB**	ED		set 5,l	
238	**INPUT**	EE	xor N	set 5,(hl)	
239	**LOAD**	EF	rst 4Ø	set 5,a	
24Ø	**LIST**	FØ	ret p	set 6,b	
241	**LET**	F1	pop af	set 6,c	
241	**PAUSE**	F2	jp p,NN	set 6,d	
243	**NEXT**	F3	di	set 6,e	
244	**POKE**	F4	call p,NN	set 6,h	
245	**PRINT**	F5	push af	set 6,l	
246	**PLOT**	F6	or N	set 6,(hl)	
247	**RUN**	F7	rst 48	set 6,a	
248	**SAVE**	F8	ret m	set 7,b	
249	**RAND**	F9	ld sp,hl	set 7,c	
25Ø	**IF**	FA	jp m,NN	set 7,d	
251	**CLS**	FB	ei	set 7,e	
252	**UNPLOT**	FC	call m,NN	set 7,h	
253	**CLEAR**	FD	prefixes instructions using iy	set 7,l	

Code	*Character*	*Hex*	*Z80 assembler*	*-after CB*	*-after ED*
254	**RETURN**	FE	cp N	set 7,(hl)	
255	**COPY**	FF	rst 56	set 7,a	

Here are the 22 graphic symbols.

Symbol	*Code*	*How obtained*	*Symbol*	*Code*	*How obtained*
□	0	K or L **SPACE**	█	128	G **SPACE**
▘	1	G shifted 1	▟	129	G shifted Q
▝	2	G shifted 2	▙	130	G shifted W
▀	3	G shifted 7	▄	131	G shifted 6
▖	4	G shifted 4	▜	132	G shifted R
▌	5	G shifted 5	▐	133	G shifted 8
▞	6	G shifted T	▚	134	G shifted Y
▛	7	G shifted E	▗	135	G shifted 3
▒	8	G shifted A	🮐	136	G shifted H
🮏	9	G shifted D	🮑	137	G shifted G
🮎	10	G shifted S	🮒	138	G shifted F

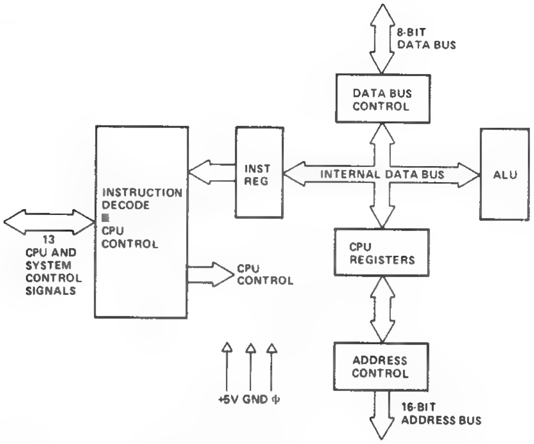
APPENDIX J
Summary of the Z80

Appendixes J and K are reprinted from *1982/1983 Z80 Designers Guide* by permission of Mostek Corporation. All figure and section numbers refer to those used in that guide.

2.0 Z80-CPU ARCHITECTURE

A block diagram of the internal architecture of the Z80-CPU is shown in Figure 2.0-1. The diagram shows all of the major elements in the CPU and it should be referred to throughout the following description.

Z80-CPU BLOCK DIAGRAM
Figure 2.0-1



2.1 CPU REGISTERS

The Z80-CPU contains 208 bits of R/W memory that are accessible to the programmer. Figure 2.0-2 illustrates how this memory is configured into eighteen 8-bit registers and four 16-bit registers. All

Z80 registers are implemented using static RAM. The registers include two sets of six general purpose registers that may be used individually as 8-bit registers or in pairs as 16-bit registers. There are also two sets of accumulator and flag registers.

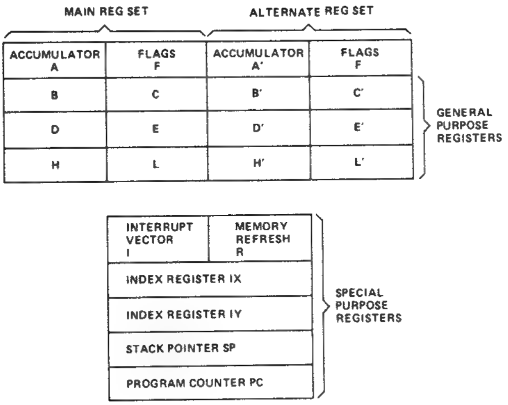
Special Purpose Registers

1. **Program Counter (PC).** The program counter holds the 16-bit address of the current instruction being fetched from memory. The PC is automatically incremented after its contents have been transferred to the address lines. When a program jump occurs, the new value is automatically placed in the PC, overriding the incrementer.

2. **Stack Pointer (SP).** The stack pointer holds the 16-bit address of the current top of a stack located anywhere in external system RAM memory. The external stack memory is organized as a last-in first-out (LIFO) file. Data can be pushed onto the stack from specific CPU registers or popped off the stack into specific CPU registers through the execution of PUSH and POP instructions. The data popped from the stack is always the last data pushed onto it. The stack allows simple implementation of multiple level interrupts, unlimited subroutine nesting and simplification of many types of data manipulation.

3. **Two Index Registers (IX & IY).** The two independent index registers hold a 16-bit base address that is used in indexed addressing modes. In this mode, an index register is used as a base to point to a region in memory from which data is to be stored or retrieved. An additional byte is included in indexed instructions to specify a displacement from this base. This displacement is specified as a two's complement signed integer. This mode of addressing greatly simplifies many types of programs, especially where tables of data are used.

Z80-CPU REGISTER CONFIGURATION
Figure 2.0-2



MAIN REG SET		ALTERNATE REG SET		
ACCUMULATOR A	FLAGS F	ACCUMULATOR A'	FLAGS F	
B	C	B'	C'	GENERAL PURPOSE REGISTERS
D	E	D'	E'	
H	L	H'	L'	

INTERRUPT VECTOR I	MEMORY REFRESH R	SPECIAL PURPOSE REGISTERS
INDEX REGISTER IX		
INDEX REGISTER IY		
STACK POINTER SP		
PROGRAM COUNTER PC		

4. **Interrupt Page Address Register (I).** The Z80-CPU can be operated in a mode where an indirect call to any memory location can be achieved in response to an interrupt. The I Register is used for this purpose to store the high order 8-bits of the indirect address while the interrupting device provides the lower 8-bits of the address. This feature allows interrupt routines to be dynamically located anywhere in memory with absolute minimal access time to the routine.

5. **Memory Refresh Register (R).** The Z80-CPU contains a memory refresh counter to enable dynamic memories to be used with the same ease as static memories. This 7-bit register is

automatically incremented after each instruction fetch. The data in the register is automatically incremented after each instruction fetch. The data in the refresh counter is sent out on the lower portion of the address bus along with a refresh control signal while the CPU is decoding and executing the fetched instruction. This mode of refresh is totally transparent to the programmer and does not slow down the CPU operation. The programmer can load the R register for testing purposes, but this register is normally not used by the programmer.

Accumulator and Flag Registers

The CPU includes two independent 8-bit accumulators and associated 8-bit flag registers. The accumulator holds the results of 8-bit arithmetic or logical operations while the flag register indicates specific conditions for 8 or 16-bit operations, such as indicating whether or not the result of an operation is equal to zero. The programmer selects with a single exchange instruction the accumulator and flag pair that he wishes to work with so that he may easily work with either pair.

General Purpose Registers

There are two matched sets of general purpose registers, each set containing six 8-bit registers that may be used individually as 8-bit register or as 16-bit register pairs by the programmer. One set is called BC, DE, and HL while the complementary set is called BD', DE' and HL'. At any one time the programmer can select either set of registers to work with through a single exchange command for the entire set. In systems where fast interrupt response is required, one set of general purpose registers and an accumulator/flag register may be reserved for handling this very fast routine. Only a simple exchange command need be executed to go between the routines. This command greatly reduces interrupt service time by eliminating the requirement for saving and retrieving register contents in the external stack during interrupt or subroutine processing. These general purpose registers are used for a wide range of applications by the programmer. They also simplify programming, especially in ROM based systems where little external read/write memory is available.

2.2 ARITHMETIC & LOGIC UNIT (ALU)

The 8-bit arithmetic and logical instructions of the CPU are executed in the ALU. Internally the ALU communicates with the registers and the external data bus on the internal data bus. The type of functions performed by the ALU includes:

Add	Left or right shifts or rotates (arithmetic and logical)
Subtract	Increment
Logical AND	Decrement
Logical OR	Set bit
Logical Exclusive OR	Reset bit
Compare	Test bit

2.3 INSTRUCTION REGISTER AND CPU CONTROL

As each instrution is fetched from memory, it is placed in the instruction register and decoded. The control section performs this function, then generates and supplies all of the control signals necessary to read or write data from or to the registers, controls the ALU and provides all required external control signals.

APPENDIX K
Machine Instructions

5.0 Z80-CPU INSTRUCTION SET

The Z80-CPU can execute 158 different instruction types including all 78 of the 8080A CPU. The instructions can be broken down into the following major groups:

- Load and Exchange
- Block Transfer and Search
- Arithmetic and Logical
- Rotate and Shift
- Bit Manipulation (set, reset, test)
- Jump, Call and Return
- Input/Output
- Basic CPU Control

5.1 INTRODUCTION TO INSTRUCTION TYPES

The load instructions move data internally between CPU registers or between CPU registers and external memory. All of these instructions must specify a source location, from which the data is to be moved, and a destination location. The source location is not altered by a load instruction. Examples of load group instructions include moves between any of the general purpose registers, such as a move of the data to Register B from Register C. This group also includes load immediate to any CPU register or to any external memory location. Other types of load instructions allow transfer between CPU registers and memory locations. The exchange instructions can trade the contents of two registers.

A unique set of block transfer instructions is provided in the Z80. With a single instruction a block of memory of any size can be moved to any other location in memory. This set of block moves is extremely valuable when large strings of data must be processed. The Z80 block search instructions are also valuable for this type of processing. With a single instruction, a block of external memory of any desired length can be searched for any 8-bit character. Once the character is found the instruction automatically terminates. Both the block transfer and the block search instructions can be interrupted during their execution so as not to occupy the CPU for long periods of time.

The arithmetic and logical instructions operate on data stored in the accumulator and other general purpose CPU registers or external memory locations. The results of the operations are placed in the accumulator and the appropriate flags are set according to the result of the operation. An example of an arithmetic operation is adding the accumulator to the contents of an external memory location. The results of the addition are placed in the accumulator. This group also includes 16-bit addition and subtraction between 16-bit CPU registers.

The bit manipulation instructions allow any bit in the accumulator, any general purpose register or any external memory location to be set, reset or tested with a single instruction. For example, the most significant bit of register H can be reset. This group is especially useful in control applications and for controlling software flags in general purpose programming.

The jump, call and return instructions are used to transfer an address between various locations in the user's program. This group uses several different techniques for obtaining the new program counter address from specific external memory locations. A unique type of jump is the restart instruction. This instruction actually contains the new address as a part of the 8-bit OP code. This type of jump is possible since only 8 separate addresses located in page zero of the external memory may be specified. Program jumps may also be achieved by loading register HL, IX or IY directly into the PC, thus allowing the jump address to be a complex function of the routine being executed.

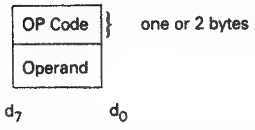
The input/output group of instructions in the Z80 allows for a wide range of transfers between external memory locations or the general purpose CPU registers, and the external I/O devices. In each case, the port number is provided on the lower 8 bits of the address bus during any I/O transaction. One instruction allows this port number to be specified by the second byte of the instruction, while other Z80 instructions allow it to be specified as the content of the C register. One major advantage of using the C register as a pointer to the I/O device is that it allows different I/O ports to share common software driver routines. This capability is not possible when the address is part of the OP code if the routines are stored in ROM. Another feature of these input instructions is that they set the flag register automatically so that additional operations are not required to determine the state of the input data (for example its parity). The Z80-CPU includes single instructions that can move blocks or data (up to 256 bytes) automatically to or from any I/O port directly to any memory location. In conjunction with the dual set of general purpose registers, these instructions provide for fast I/O block transfer rates. The value of this I/O instruction set is demonstrated by the fact that the Z80-CPU can provide all required floppy disk formatting (i.e., the CPU provides the preamble, address, and data, and enables the CRC codes) on double density floppy disk drives on an interrupt driven basis.

Finally, the basic CPU control instructions allow various options and modes. This group includes instructions such as setting or resetting the interrupt enable flip flop or setting the mode of interrupt response.

5.2 ADDRESSING MODES

Most of the Z80 instructions operate on data stored in the internal CPU registers, the external memory, or in the I/O ports. Addressing refers to how the address of this data is generated in each instruction. This section gives a brief summary of the types of addressing used in the Z80 while subsequent sections detail the type of addressing available for each instruction group.

Immediate. In this mode of addressing, the byte following the OP code in memory contains the actual operand.



An example of this type of instruction would be to load the HL register pair (16-bit register) with 16 bits (2 bytes) of data.

Immediate Extended. This mode is merely an extension of immediate addressing, in that the two bytes following the op codes are the operand.

OP Code	one or 2 bytes
Operand	low order
Operand	high order

An example of this type of instruction would be to load the HL register pair (16-bit register) with 16 bits (2 bytes) of data.

Modified Page Zero Addressing. The Z80 has a special single byte call instruction to any of 8 locations in page zero of memory. This instruction (which is referred to as a restart) sets the PC to an effective address in page zero. The value of this instruction is that it allows a single byte to specify a complete 16-bit address where commonly called subroutines are located, thus saving memory space.

OP Code	one byte

b_7 d_0 Effective address is ($00b_5b_4b_3000$)

Relative Addressing. Relative addressing uses one byte of data following the OP code to specify a displacement from the existing program to which a program jump can occur. This displacement is a signed two's complement number that is added to the address of the OP code of the following instruction.

OP Code	Jump relative (one byte OP code)
Operand	8-bit two's complement displacement added to Address (A+2)

The value of relative addressing is that it allows jumps to nearby locations while only requiring two bytes of memory space. For most programs, relative jumps are by far the most prevalent type of jump owing to the proximity of related program segments. Thus, these instructions can significantly reduce memory space requirements. The signed displacement can range between +127 and −128 from A + 2. This allows for a total displacement of +129 to −126 from the jump relative OP code address. Another major advantage is that it allows for relocatable code.

Extended Addressing. Extended Addressing provides for two bytes (16 bits) of address to be included in the instruction. This data can be an address to which a program can jump or it can be an address where an operand is located.

OP Code	one or two bytes
Low Order Address or Low order operand	
High Order Address or High order operand	

Extended addressing is required for a program to jump from any location in memory to any other location, or load and store data in any memory location.

When extended addressing is used to specify the source or destination address of an operand, the notation (nn) will be used to indicate the content of memory at nn, where nn is the 16-bit address specified in the instruction. This means that the two bytes of address nn are used as a pointer to a memory location. The use of the parentheses always means that the value enclosed within them is used as a pointer to a memory location. For example, (1200) refers to the contents of memory at location 1200.

Indexed Addressing. In this type of addressing, the byte of data following the OP code contains a displacement which is added to one of the two index registers (the OP code specifies which index register is used) to form a pointer to memory. The contents of the index register are not altered by this operation.

OP Code	two byte OP code
OP Code	
Displacement	Operand added to index register to form a pointer to memory.

An example of an index instruction would be to load the contents of the memory location (Index Register + Displacement) into the accumulator. The displacement is a signed two's complement number. Indexed addressing greatly simplifies programs using tables of data since the index register can point to the start of any table. Two index registers are provided since, very often, operations require two or more tables. Indexed addressing also allows for relocatable code.

The two index registers in the Z80 are referred to as IX and IY. To indicate indexed addressing, the notation:

(IX+d) or (IY+d)

is used. Here d is the displacement specified after the OP code. The parentheses indicate that this value is used as a pointer to external memory.

Register Addressing. Many of the Z80 OP codes contain bits of information that specify which CPU register is to be used for an operation. An example of register addressing would be to load the data in register B into register C.

Implied Addressing. Implied addressing refers to operations where the OP code automatically implies one or more CPU registers as containing the operands. An example is the set of arithmetic operations where the accumulator is always implied to be the destination of the results.

Register Indirect Addressing. This type of addressing specifies a 16-bit CPU register pair (such as HL) to be used as a pointer to any location in memory. This type of instruction is very powerful and it is used in a wide range of applications.

OP Code	one or two bytes

An example of this type of instruction would be to load the accumulator with the data in the memory location pointed to by the HL register contents. Indexed addressing is actually a form of register indirect addressing except that a displacement is added with indexed addressing. Register indirect addressing allows for very powerful but simple to implement memory accesses. The block move and search commands in the Z80 are extensions of this type of addressing where automatic register incrementing, decrementing and comparing have been added. The notation for indicating register indirect addressing is to put parentheses around the name of the register that is to be used as the pointer. For example, the symbol

(HL)

specifies that the contents of the HL register are to be used as a pointer to a memory location. Often register indirect addressing is used to specify 16-bit operands. In this case, the register contents point to the lower order portion of the operand while the register contents are automatically incremented to obtain the upper portion of the operand.

Bit Addressing. The Z80 contains a large number of bit set, reset and test instructions. These instructions allow any memory location or CPU register to be specified for a bit operation through one of three previous addressing modes (register, register indirect and indexed), while three bits in the OP code specify which of the eight bits is to be manipulated.

ADDRESSING MODE COMBINATIONS

Many instructions include more than one operand (such as arithmetic instructions or loads). In these cases, two types of addressing may be employed. For example, load can use immediate addressing to specify the source, and register indirect or indexed addressing to specify the source, and register indirect or indexed addressing to specify the destination.

5.3 INSTRUCTION OP CODES

This section describes each of the Z80 instructions and provides tables listing the OP codes for every instruction. In each of these tables, the shaded OP codes are identical to those offered in the 8080A CPU. Also shown is the assembly language mnemonic that is used for each instruction. All instruction OP codes are listed in hexadecimal notation. Single byte OP codes require two hex characters while double byte OP codes require four hex characters. The conversion from hex to binary is repeated here for convenience.

Hex	Binary	Decimal	Hex	Binary	Decimal
0	= 0000 =	0	8	= 1000 =	8
1	= 0001 =	1	9	= 1001 =	9
2	= 0010 =	2	A	= 1010 =	10
3	= 0011 =	3	B	= 1011 =	11
4	= 0100 =	4	C	= 1100 =	12
5	= 0101 =	5	D	= 1101 =	13
6	= 0110 =	6	E	= 1110 =	14
7	= 0111 =	7	F	= 1111 =	15

Z80 instruction mnemonics consist of an OP code and zero, onė or two operands. Instructions in which the operand is implied have no operand. Instructions which have only one logical operand or those in which one operand is invariant (such as the Logical OR instruction) are represented by a one operand mnemonic. Instructions which may have two varying operands are represented by two operand mnemonics.

LOAD AND EXCHANGE

Table 5.3-1 defines the OP code for all of the 8-bit load instructions implemented in the Z80-CPU. Also shown in this table is the type of addressing used for each instruction. The source of the data is found on the top horizontal row while the destination is specified by the left hand column. For example, load register C from register B uses the OP code 48H. In all of the tables the OP code is specified in hexadecimal notation and the 48H (=0100 1000 binary) code is fetched by the CPU from the external memory during M1 time, is decoded and then the register transfer is automatically performed by the CPU.

The assembly language mnemonic for this entire group is LD, followed by the destination, followed by the source (LD DEST., SOURCE). Note that several combinations of addressing modes are possible. For example, the source may use register addressing and the destination may be register indirect, as in the case of loading the memory location pointed to by register HL with the contents of register D. The OP code for this operation would be 72. The mnemonic for this load instruction would be as follows: LD (HL), D

The parentheses around the HL mean that the contents of HL are used as a pointer to a memory location. In all Z80 load instruction mnemonics, the destination is always listed first, with the source following. The Z80 assembly language has been defined for ease of programming. Every instruction is self documenting and programs written in Z80 language are easy to maintain.

Note in Table 5.3-1 that some load OP codes that are available in the Z80 use two bytes. This is an efficient method of memory utilization, since 8, 16, 24 or 32 bit instructions are implemented in the

Z80. Thus often utilized instructions such as arithmetic or logical operations are only 8-bits which result in better memory utilization than is achieved with fixed instruction sizes such as 16-bits.

All load instructions using indexed addressing for either the source or destination location actually use three bytes of memory with the third byte being the displacement d. For example, a load register E, with the operand pointed to by IX with an offset of +8, would be written: LD E, (IX + 8).

The instruction sequence for this in memory would be:

Address A	DD	OP Code
A+1	5F	
A+2	08	Displacement operand

The two extended addressing instructions are also three byte instructions. For example the instruction to load the accumulator with the operand in memory location 6F32H would be written as:

LD A, (6F 32H)

and its instruction sequence would be:

Address A	3A	OP Code
A+1	32	low order address
A+2	6F	high order address

Notice that the low order portion of the address is always the first operand.

The load immediate instructions for the general purpose 8-bit registers are two-byte instructions. The instruction load register H with the value 36H would be written as:

LD H, 36H

and its sequence would be:

Address A	26	OP Code
.+1	36	Operand

Loading a memory location using indexed addressing for the destination and immediate addressing for the source requires four bytes. For example,

LD (IX - 15), 21H

would appear as:

Address A	DD	OP Code
A+1	36	
A+2	F1	displacement (–15 in signed two's complement)
A+3	21	operand to load

Notice that with any indexed addressing the displacement always follows directly after the OP code.

Table 5.3-2 specifies the 16-bit load operations. This table is very similar to Table 5.3-1. Notice that the extended addressing capability covers all register pairs. Also notice that register indirect operations specifying the stack pointer are the PUSH and POP Instructions. The mnemonics for these instructions are "PUSH" and "POP". These instructions differ from other 16-bit loads in that the stack pointer is automatically decremented and incremented as each byte is pushed onto or popped from the stack respectively. For example, the instruction

PUSH AF

is a single byte instruction with the OP code of F5H. When this instruction is executed the following sequence is generated:

```
Decrement SP

LD (SP), A

Decrement SP

LD (SP), F
```

Thus the external stack now appears as follows:

(SP)	F	Top of stack
(SP+1)	A	
•	•	
•	•	

8 BIT LOAD GROUP
Table 5.3-1

SOURCE

		IMPLIED		REGISTER							REG INDIRECT			INDEXED		EXT. ADDR.	IMME.
DESTINATION		I	R	A	B	C	D	E	H	L	(HL)	(BC)	(DE)	(IX + d)	(IY + d)	(nn)	n
REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
EXT ADDR	(nn)			32 n n													
IMPLIED	I			ED 47													
	R			ED 4F													

The POP instruction is the exact reverse of a PUSH. Notice that all PUSH and POP instructions utilize a 16-bit operand and the high order byte is always pushed first and popped last.
That is:

PUSH BC is PUSH B then C
PUSH DE is PUSH D then E
PUSH HL is PUSH H then L
POP HL is POP L then H

The instruction using extended immediate addressing for the source obviously requires 2 bytes of data following the OP code. For example:

LD DE, 0659H

will be:

Address A	11	OP Code
A+1	59	Low order operand to register E
A+2	06	High order operand to register D

In all extended immediate or extended addressing modes, the low order byte always appears first after the OP code.

Table 5.3-3 lists the 16-bit exchange instructions implemented in the Z80. OP code 08H allows the programmer to switch between the two pairs of accumulator flag registers while D9H allows the programmer to switch between the duplicate set or six general purpose registers. These OP codes are only one byte in length to minimize the time necessary to perform the exchange absolutely, so that the duplicate banks can be used to effect very fast interrupt response times.

BLOCK TRANSFER AND SEARCH

Table 5.3-4 lists the extremely powerful block transfer instructions. All of these instructions operate with three registers:

HL points to the source location
DE points to the destination location.
BC is a byte counter.

After the programmer has initialized these three registers, any of these four instructions may be used. The LDI (Load and Increment) instruction moves one byte from the location pointed to by HL to the location pointed to by DE. Register pairs HL and DE are then automatically incremented and are ready to point to the following locations. The byte counter (register pair BC) is also decremented at this time. This instruction is valuable when blocks of data must be moved, but other types of processing are required between each move. The LDIR (load, increment and repeat) instruction is an extension of the LDI instruction. The same load and increment operation is repeated until the byte counter reaches the count of zero. Thus, this single instruction can move any block of data from one location to any other.

Note that since 16-bit registers are used, the size of the block can be up to 64K bytes (1K = 1024) long, and it can be moved from any location in memory to any other location. Furthermore the blocks can be overlapping since there are absolutely no constraints on the data that is used in the three register pair.

The LDD and LDDR instructions are very similar to the LDI and LDIR. The only difference is that register pairs HL and DE are decremented after every move so that a block transfer starts from the highest address of the designated block rather than the lowest.

BIT LOAD GROUP 'LD' 'PUSH' and 'POP'
Table 5.3.-2

											SOURCE
		REGISTER							IMM. EXT.	EXT. ADDR.	REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
DESTINATION REGISTER	AF										F1
	BC								[illegible]	ED 4B n n	[illegible]
	DE								[illegible]	ED 5B n n	[illegible]
	HL								[illegible]	[illegible]	E1
	SP				F9		DD F9	FD F9	[illegible]	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	[illegible]	ED 73 n n	DD 22 n n	FD 22 n n			
PUSH INSTRUCTIONS → REG. IND.	(SP)	[illegible]	[illegible]	[illegible]	[illegible]		DD E5	FD E5			

NOTE: The Push & Pop Instructions adjust the SP after every execution

↑ POP INSTRUCTIONS

CHANGES 'EX' and 'EXX'
Table 5.3-3

		IMPLIED ADDRESSING				
		AF'	BC', DE' & HL'	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			[illegible]		
REG. INDIR.	(SP)			[illegible]	DD E3	FD E3

BLOCK TRANSFER GROUP
Table 5.3-4

			SOURCE REG. INDIR. (HL)	
DESTINATION	REG. INDIR.	(DE)	ED A0	'LDI' – Load (DE)←—(HL) Inc HL & DE, Dec BC
			ED B0	'LDIR,' – Load (DE)←—(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
			ED A8	'LDD' – Load (DE)←—(HL) Dec HL & DE, Dec BC
			ED 88	'LDDR' – Load (DE)←—(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Reg	HL	points to source
Reg	DE	points to destination
Reg	BC	is byte counter

Table 5.3-5 specifies the OP codes for the four block search instructions. The first, CPI (compare and increment) compares the data in the accumulator, with the contents of the memory location pointed to by register HL. The result of the compare instruction is stored in one of the flag bits (see section 6.0 for a detailed explanation of the flag operations) and the HL register pair is then incremented and the byte counter (register pair BC) is decremented.

The instruction CPIR is merely an extension of the CPI instruction in which the compare is repeated until either a match is found or the byte counter (register pair BC) becomes zero. Thus, this single instruction can search the entire memory for any 8-bit character.

The CPD (Compare and Decrement) and CPDR (Compare, Decrement and Repeat) are similar instructions, their only difference being that they decrement HL after every compare instruction so that they search the memory in the opposite direction. (The search is started at the highest location in the memory block).

It should be emphasized again that these block transfer and compare instructions are extremely powerful in string manipulation applications.

ARITHMETIC AND LOGICAL

Table 5.3-6 lists all of the 8-bit arithmetic operations that can be performed with the accumulator; also listed are the increment (INC) and decrement (DEC) instructions. In all of these instructions, except INC and DEC, the specified 8-bit operation is performed between the data in the accumulator and the source data specified in the table. The result of the operation is placed in the accumulator with the exception of compare (CP) that leaves the accumulator unaffected. All of these operations affect the flag register as a result of the specified operation. (Section 6.0 provides all of the details on how the flags are affected by any instruction type). INC and DEC instructions specify a register or a memory location as both source and destination of the result. When the source operand is addressed using the index registers, the displacement must follow directly. With immediate addressing the actual operand will follow directly. For example, the instruction

AND 07H

would appear as:

Address A	E6	OP Code
A+1	07	Operand

BLOCK SEARCH GROUP

Table 5.3-5

SEARCH LOCATION REG. INDIR. (HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repeat until BC = 0 or find match

HL points to location in memory
to be compared with accumulator
contents
BC is byte counter

Assuming that the accumulator contained the value F3H, the result of 03H would be placed in the accumulator:

Acc before operation	1111 0011 = F3H
Operand	0000 0111 = 07H
Result to Acc	0000 0011 = 03H

The Add instruction (ADD) performs a binary add between the data in the source location and the data in the accumulator. The subtract (SUB) does a binary subtraction. When the add with carry is specified (ADC) or the subtract with carry (SBC), then the carry flag is also added or subtracted respectively. The flags and decimal adjust instruction (DAA) in the Z80 (fully described in section 6.0) allow arithmetic operations for:

multiprecision packed BCD numbers

multiprecision signed or unsigned binary numbers

multiprecision two's complement signed numbers

Other instructions in this group are: logical and (AND); logical or (OR); exclusive or (XOR), and compare (CP).

There are five general purpose arithmetic instructions that operate on the accumulator or carry flag. These five are listed in Table 5.3-7. The decimal adjust instruction can adjust for subtraction as well as addition, thus making BCD arithmetic operations simple. Note that to allow for this operation, the flag N is used. This flag is set if the last arithmetic operation was a subtract. The negate accumulator (NEG) instruction forms the two's complement of the number in the accumulator. Finally, notice that a reset carry instruction is not included in the Z80 since this operation can be easily achieved through other instructions such as a logical AND of the accumulator with itself.

Table 5.3-8 lists all of the 16-bit arithmetic operations between 16-bit registers. There are five groups of instructions, including add with carry and subtract with carry. ADC and SBC affect all of the flags. These two groups simplify address calculation operations or other 16-bit arithmetic operations.

8 BIT ARITHMETIC AND LOGIC

Table 5.3-6

SOURCE

	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

GENERAL PURPOSE AF OPERATIONS

Table 5.3-7

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

16 BIT ARITHMETIC
Table 5.3-8

			SOURCE					
			BC	DE	HL	SP	IX	IY
DESTINATION	'ADD'	HL	09	19	29	39		
		IX	DD 09	DD 19		DD 39	DD 29	
		IY	FD 09	FD 19		FD 39		FD 29
	ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
	SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
	INCREMENT 'INC.		03	13	23	33	DD 23	FD 23
	DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

ROTATE AND SHIFT

A major capability of the Z80 is its ability to rotate or shift data in the accumulator, any general purpose register, or any memory location. All of the rotate and shift OP codes are shown in Table 5.3-9. Also included in the Z80 are arithmetic and logical shift operations. These operations are useful in an extremely wide range of applications including integer multiplication and division. Two BCD digit rotate instructions (RRD and RLD) allow a digit in the accumulator to be rotated with the two digits in a memory location pointed to by register pair HL. (See Figure 5.3-9). These instructions allow for efficient BCD arithmetic.

BIT MANIPULATION

The ability to set, reset, and test individual bits in a register or memory location is needed in almost every program. These bits may be flags in a general purpose software routine, may be indications of external control conditions, or may be data packed into memory locations to make memory utilization more efficient.

The Z80 has the ability to set, reset, or test any bit in the accumulator, any general purpose register or any memory location with a single instruction. Table 5.3-10 lists the 240 instructions that are available for this purpose. Register addressing can specify the accumulator or any general purpose register on which the operation is to be performed. Register indirect and indexed addressing are available to operate on external memory locations. Bit test operations set the zero flag (Z) if the tested bit is a zero. (Refer to section 6.0 for further explanation of flag operation).

JUMP, CALL, AND RETURN

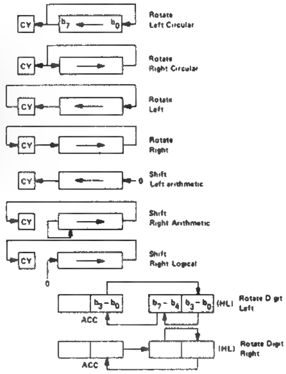
Figure 5.3-11 lists all of the jump, call and return instructions implemented in the Z80 CPU. A jump is a branch in a program where the program counter is loaded with the 16-bit value as specified by one of the three available addressing modes (Immediate Extended, Relative, or Register Indirect). Notice that the jump group has several different conditions that can be specified to be met before the jump will be made. If these conditions are not met, the program merely continues with the next sequential instruction. The conditions are all dependent on the data in the flag register. (Refer to section 6.0 for details on the flag register). The immediate extended addressing is used to jump to any location in the memory. This instruction requires three bytes (two to specify the 16-bit address) with the low order address byte first followed by the high order address byte.

ROTATES AND SHIFTS

Table 5.3-9

		Source and Destination									
		A	B	C	D	E	H	L	(HL)	(IX + d)	(IY + d)
TYPE OF ROTATE OR SHIFT	'RLC'	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB d 06	FD CB d 06
	'RRC'	CB 0F	CB 08	CB 09	CB 0A	CB 0B	CB 0C	CB 0D	CB 0E	DD CB d 0E	FD CB d 0E
	'RL'	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB d 16	FD CB d 16
	'RR'	CB 1F	CB 18	CB 19	CB 1A	CB 1B	CB 1C	CB 1D	CB 1E	DD CB d 1E	FD CB d 1E
	'SLA'	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB d 26	FD CB d 26
	'SRA'	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB d 2E	FD CB d 2E
	'SRL'	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB d 3E	FD CB d 3E
	'RLD'								ED 6F		
	'RRD'								ED 67		

	A
RLCA	07
RRCA	0F
RLA	17
RRA	1F



For example an unconditional Jump to memory location 3E32H would be:

Address A	C3	OP Code
A+1	32	Low order address
A+2	3E	High order address

The relative jump instruction uses only two bytes; the second byte is a signed two's complement displacement from the existing PC. This displacement can be in the range of +129 to -126 and is measured from the address of the instruction OP code.

Three types of register indirect jumps are also included. These instructions are implemented by loading the register pair HL or one of the index registers IX or IY directly into the PC. This capability allows for program jumps to be a function of previous calculations.

A call is a special form of a jump where the address of the byte following the call instruction is pushed onto the stack before the jump is made. A return instruction is the reverse of a call because the data on the top of the stack is popped directly into the PC to form a jump address. The call and return instructions allow for simple subroutine and interrupt handling. Two special return instructions have been included in the Z80 family of components. The return from interrupt instruction (RETI) and the return from non-maskable interrupt (RETN) are treated in the CPU as an unconditional return identical to the OP code C9H. The difference is that (RETI) can be used at the end of an interrupt routine and all Z80 peripheral chips will recognize the execution of this instruction for proper control of nested priority interrupt handling. This instruction coupled with the Z80 peripheral devices' implementation simplifies the normal return from nested interrupt. Without this feature, the following software sequence would be necessary to inform the interrupting device that the interrupt routine has been completed:

BIT MANIPULATION GROUP

Table 5.3-10

		REGISTER ADDRESSING							REG INDIR.	INDEXED	
	BIT	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
TEST 'BIT'	0	CB 47	C[illegible] [illegible]	[illegible] [illegible]1	C[illegible] [illegible]2	CB 4[illegible]	CB 44	CB 45	CB 46	DD CB d 46	FD CB d 46
	1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB d 4E	FD CB d 4E
	2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB d 56	FD CB d 56
	3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB d 5E	FD CB d 5E
	4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB d 66	FD CB d 66
	5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB d 6E	FD CB d 6E
	6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	6B 75	CB 76	DD CB d /6	FD CB d 76
	7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB d 7E	FD CB d 7E
RESET BIT 'RES'	0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB d 86	FD CB d 86
	1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB d 8E	FD CB d 8E
	2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB d 96	FD CB d 96
	3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB d 9E	FD CB d 9E
	4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB d A6	FD CB d A6
	5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB d AE	FD CB d AE
	6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB d B6	FD CB d B6
	7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB d BE	FD CB d BE
SET BIT 'SET'	0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB d C6	FD CB d C6
	1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB d CE	FD CB d CE
	2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB d D6	FD CB d D6
	3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB d DE	FD CB d DE
	4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB d E6	FD CB d E6
	5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB d EE	FD [illegible] d EE
	6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB d F6	FD CB d F6
	7	CB FF	CB F8	CB F9	CB FA	CB FB	CB FC	CB FD	CB FE	DD CB d FE	FD CB d FE

Disable Interrupt	— prevent interrupt before routine is exited.
LD A,n OUT n, A	— notify peripheral that service routine is complete
Enable Interrupt	
Return	

This seven byte sequence can be replaced with the three byte EI RETI instruction sequence in the Z80. This is important since interrupt service time often must be minimized.

To facilitate program loop control, the instruction DJNZ e can be used advantageously. This two byte, relative jump instruction decrements the B register, and the jump occurs if the B register has not been decremented to zero. The relative displacement is expressed as a signed two's complement number. A simple example of its use might be:

Address	Instruction	Comments
N,N+1	LD B,7	; set B register to count of 7
N+2 to N+9	(Perform a sequence of instructions)	; loop to be performed 7 times
N+10, N+11	DJNZ -10	; to jump from N + 12 to N + 2
N + 12	(Next Instruction)	

JUMP, CALL, AND RETURN GROUP

Table 5.3-11

			CONDITION									
			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B≠0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

NOTE—CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO SECTION 6.0 FOR DETAILS

Table 5.3-12 lists the eight OP codes for the restart instruction. This instruction is a single byte call to any of the eight addresses listed. The simple mnemonic for these eight calls is also shown. The value of this instruction is that frequently used routines can be called with this instruction to minimize memory usage.

RESTART GROUP
Table 5.3-12

		OP CODE	
CALL ADDRESS	0000_H	C7	'RST 0'
	0008_H	CF	'RST 8'
	0010_H	D7	'RST 16'
	0018_H	DF	'RST 24'
	0020_H	E7	'RST 32'
	0028_H	EF	'RST 40'
	0030_H	F7	'RST 48'
	0038_H	FF	'RST 56'

INPUT/OUTPUT

The Z80 has an extensive set of Input and Output instructions, as shown in table 5.3-13 and table 5.3-14. The addressing of the input or output device can be either absolute or register indirect, using the C register. Notice that in the register indirect addressing mode, data can be transferred between the I/O devices and any of the internal registers. In addition eight block transfer instructions have been implemented. These instructions are similar to the memory block transfers except that they use register pair HL for a pointer to the memory source (output commands) or destination (input commands), while register B is used as a byte counter. Register C holds the address of the port for which the input or output command is desired. Since register B is eight bits in length, the I/O block transfer command handles up to 256 bytes.

In the instructions IN A, n and OUT n, A, an I/O device address n appears in the lower half of the address bus (A_0-A_7) while the accumulator content is transferred in the upper half of the address bus. In all register indirect input output instructions, including block I/O transfers, the content of register C is transferred to the lower half of the address bus (device address) while the content of register B is transferred to the upper half of the address bus.

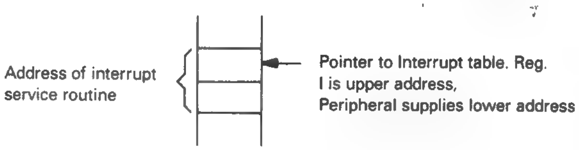
INPUT GROUP

Table 5.3-13

				PORT ADDRESS		
				IMMED.	REG. INDIR.	
				n	(C)	
INPUT DESTINATION	INPUT 'IN'	REG ADDRESSING	A	[illegible]	ED 78	
			B		ED 40	
			C		ED 48	
			D		ED 50	
			E		ED 58	
			H		ED 60	
			L		ED 68	
	'INI' – INPUT & Inc HL, Dec B	REG, INDIR	(HL)		ED A2	BLOCK INPUT COMMANDS
	'INIR'– INP, Inc HL, Dec B, REPEAT IF B≠0				ED B2	
	'IND'– INPUT & Dec HL, Dec B				ED AA	
	'INDR'– INPUT, Dec HL, Dec B, REPEAT IF B≠0				ED BA	

CPU CONTROL GROUP

The final table, table 5.3-15, illustrates the six general purpose CPU control instructions. The NOP is a do-nothing instruction. The HALT instruction suspends CPU operation until a subsequent interrupt is received, while the DI and EI are used to lock out and enable interrupts. The three interrupt mode commands set the CPU into any of the three available interrupt response modes as follows. If mode zero is set, the interrupting device can insert any instruction on the data bus and allow the CPU to execute it. Mode 1 is a simplified mode where the CPU automatically executes a restart (RST) to location 0038H so that no external hardware is required. (The old PC content is pushed onto the stack). Mode 2 is the most powerful in that it allows for an indirect call to any location in memory. With this mode, the CPU forms a 16-bit memory address where the upper 8-bits are the content of mode, the CPU forms a 16-bit memory address where the upper 8-bits are the content of register I, and the lower 8-bits are supplied by the interrupting device. This address points to the first of two sequential bytes in a table where the address of the service routine is located. The CPU automatically obtains the starting address and performs a CALL to this address.


Address of interrupt
service routine
Pointer to Interrupt table. Reg.
I is upper address,
Peripheral supplies lower address

OUTPUT GROUP

Table 5.3-14

			SOURCE							
			REGISTER							REG. IND.
			A	B	C	D	E	H	L	(HL)
'OUT'	IMMED.	n	D3 n							
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB

BLOCK OUTPUT COMMANDS

PORT DESTINATION ADDRESS

MISCELLANEOUS CPU CONTROL

Table 5.3-15

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080A MODE
SET INT MODE 1 'IM1'	ED 56	CALL TO LOCATION 0038_H
SET INT MODE 2 'IM2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

6.0 FLAGS

Each of the two Z80-CPU Flag registers contains six bits of information which are set or reset by various CPU operations. Four of these bits are testable; that is, they are used as conditions for jump, call, or return instructions. For example, a jump may be desired only if a specific bit in the flag register is set. The four testable flag bits are:

1) Carry Flag (C) — This flag is the carry from the highest order bit of the accumulator. For example, the carry flag will be set during an add instruction where a carry from the highest bit of the accumulator is generated. This flag is also set if a borrow is generated during a subtraction instruction. The shift and rotate instructions also affect this bit.

2) Zero Flag (Z) — This flag is set if the result of the operation loaded a zero into the accumulator. Otherwise the flag is reset.

3) Sign Flag (S) — This flag is intended to be used with signed numbers, and it is set if the result of the operation was negative. Since bit 7 (MSB) represents the sign of the number (A negative number has a 1 in bit 7), this flag stores the state of bit 7 in the accumulator.

4) Parity/Overflow Flag (P/V) — This dual purpose flag indicates the parity of the result in the accumulator when logical operations are performed (such as AND A, B) and it represents overflow when signed two's complement arithmetic operations are performed. The Z80 overflow flag indicates that the two's complement number in the accumulator is in error since it has exceeded the maximum possible (+127) or is less than the minimum possible (-128) number that can be represented by two's complement notation. For example consider adding:

```
+120 =    0111 1000
+105 =    0110 1001
-----     ---------
   C = 0  1110 0001 = -95 (wrong) Overflow has occurred
```

Here the result is incorrect. Overflow has occurred and yet there is no carry to indicate an error. For this case, the overflow flag would be set. Also consider the addition of two negative numbers.

```
  -5 =    1111 1011
 -16 =    1111 0000
 ---      ---------
   C = 1  1110 1011 = -21 correct
```

Notice that the answer is correct, but the carry is set so that this flag cannot be used as an overflow indicator. In this case, the overflow would not be set.

For logical operations (AND, OR, XOR), this flag is set if the parity of the result is even, and the flag is reset if it is odd.

There are also two non-testable bits in the flag register. Both of these are used for BCD arithmetic. They are:

1) Half carry (H) — This is the BCD carry or borrow result from the least significant four bits of operation. When using the DAA (Decimal Adjust Instruction) this flag is used to correct the result of a previous packed decimal add or subtract.

2) Add/Subtract Flag (N) — Since the algorithm for correcting BCD operations if different for addition or subtraction, this flag is used to specify what type of instruction was executed last so that the DAA operation will be correct for either addition or subtraction.

The Flag register can be accessed by the programmer, and its form is as follows:

D7							D0
S	Z	X	H	X	P/V	N	C

Table 6.0-1 lists how each flag bit is affected by various CPU instructions. In this table, '•' indicates that the instruction does not change the flag; an 'X' means that the flag goes to an indeterminate state; an '0' means that it is reset; a '1' means that it is set, and the symbol ↕ indicates that it is set for reset according to the previous discussion. Note that any instruction not appearing in this table does not affect any of the flags.

Table 6.0-1 includes a few special cases that must be described for clarity. Notice that the block search instruction sets the Z flag if the last compare operation indicated a match between the source and the accumulator data. Also, the parity flag is set if the byte counter (register pair BC) is not equal to zero. This same use of the parity flag is made with the block move instructions. Another special case is during block input or output instructions. Here the Z flag is used to indicate the state of register B which is used as a byte counter. Notice that when the I/O block transfer is complete, the zero flag will be reset to a zero (i.e. B=0), while in the case of a block move command, the parity flag is reset when the operation is complete. A final case occurs when the refresh or I register is loaded into the accumulator, because interrupt enable flip flop is then loaded into the parity flag so that the complete state of the CPU can be saved at any time.

SUMMARY OF FLAG OPERATION

Table 6.0-1

	D7					P/		D0	
Instruction	S	Z		H		V	N	C	Comments
ADD A,s; ADC A,s	↕	↕	X	↕	X	V	0	↕	8-bit add or add with carry
SUB,s; SBCA,s; CP,s; NEG	↕	↕	X	↕	X	V	1	↕	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	↕	↕	X	1	X	P	0	0	} Logical operations
OR s; XOR s	↕	↕	X	0	X	P	0	0	
INC s	↕	↕	X	↕	X	V	0	●	8-bit increment
DEC s	↕	↕	X	↕	X	V	1	●	8-bit decrement
ADD DD, SS	●	●	X	X	X	●	0	↕	16-bit add
ADC HL, SS	↕	↕	X	X	X	V	0	↕	16-bit add with carry
SBC HL, SS	↕	↕	X	X	X	V	1	↕	16-bit subtract with carry
RLA; RLCA; RRA; RRCA	●	●	X	0	X	●	0	↕	Rotate accumulator
RL s; RLC s; RR s; RRC s; SLA s; SRA s; SRL s	↕	↕	X	0	X	P	0	↕	Rotate and shift locations
RLD; RRD	↕	↕	X	0	X	P	0	●	Rotate digit left and right
DAA	↕	↕	X	↕	X	P	●	↕	Decimal adjust accumulator
CPL	●	●	X	1	X	●	1	●	Complement accumulator
SCF	●	●	X	0	X	●	0	1	Set carry
CCF	●	●	X	X	X	●	0	↕	Complement carry
IN r, (C)	↕	↕	X	0	X	P	0	●	Input register indirect
INI; IND; OUTI; OUTD	X	↕	X	X	X	X	1	X	} Block input and output
INIR; INDR; OTIR; OTDR	X	1	X	X	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1; if bit 7 = 1, N = 1
LDI; LDD	X	X	X	0	X	↕	0	●	} Block transfer instructions
LDIR; LDDR	X	X	X	0	X	0	0	●	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI; CPIR; CPD; CPDR	↕	↕	\	X	X	↕	1	●	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	↕	↕	X	0	X	IFF	0	●	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	X	↕	X	1	X	X	0	●	The state of bit b of location s is copied into the Z flag

The following notation is used in this table:

SYMBOL	OPERATION
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result, while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V= if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract. H and N flags are used in conjunction with the decimal adjust instruction (DAA) to correct properly the result into packed BCD format following addition or subtraction using operands with packed BCD format. The flag is affected according to the result of the operation.
●	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care".
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction.
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>

Corrections to *Timex/Sinclair User's Guide, Volume 1*

	Replace	**With**
page 163	100 I=I/PER/100	100 **LET** I=I/PER/100
	to compute interest, change line 150 back to 150 **LET** P=P+IN	and delete line 175.

page 164 **Change** all references to **line 165 to line 155**.

page 174 **Add** the following lines:

403 **SCROLL**

405 **PRINT** "TOTAL PAID=";PRI+IN

INDEX TO VOLUME 1

INDEX

Que Timex/Sinclair 1000 Books

Timex/Sinclair 1000 User's Guide Volume 1	currently available
Timex/Sinclair 1000 User's Guide Volume 2	(this book)
Timex/Sinclair 1000 Pocket Dictionary	available: March, 1983
Timex/Sinclair 1000 Games	available: May, 1983

These books can be purchased at many fine computer, book, and department stores nationwide. If they are not available in your area, they can be purchased in complete sets only from Que Corporation.

Que Microcomputer Products

BOOKS:	**ISBN No.**	**Date Available**
Apple II Word Processing	0-88022-005-8	Currently
C Programming Guide	0-88022-022-8	Spring, '83
CP/M Compatible Software Catalog 2nd Ed.	0-88022-018-X	Currently
CP/M Word Processing	0-88022-006-6	Currently
IBM PC Expansion & Software Guide	0-88022-019-8	Currently
IBM PC Pocket Dictionary	0-88022-024-4	Spring, '83
IBM's Personal Computer-hbk.	0-88022-101-1	Currently
IBM's Personal Computer-pbk.	0-88022-100-3	Currently
The Osborne Portable Computer	0-88022-015-5	Currently
Personal Computers for Managers	0-88022-031-7	May, '83
SuperCalc SuperModels for Business	0-88022-007-4	Currently
Timex/Sinclair 1000 User's Guide, Vol. 1	0-88022-016-3	Currently
Timex/Sinclair 1000 Pocket Dictionary	0-88022-028-7	Spring, '83
VisiCalc Models for Business	0-88022-017-1	March, '83

SOFTWARE:

CalcSheets for Business	1100 Series	Currently

"CalcSheets for Business" is a series of VisiCalc and SuperCalc models to assist businesspeople in cash management, debt management, fixed asset management, working capital management, and other business management. These models run on the IBM Personal Computer, Apple II computer and other popular personal computers.

Notes

Notes

Notes

Notes

This book is the second volume of Que's best-selling *Timex/Sinclair 1000 User's Guide*. In Volume 2 Dr. Giarratano introduces the reader to advanced BASIC programming techniques, such as sorting, renumbering programs, plotting and drawing, computer animation and graphics, and video games.

The reader is told how to use machine-language programs for greater execution speed. There is even discussion of new hardware and software products for the Timex/Sinclair 1000.

Every person interested in this affordable computer should own this practical, informative volume on advanced programming techniques.

0 29236 01295 00295

que

0-88022-029-